# Database Systems

# Lecture 9:  XML

**Dr. Momtazi**
**momtazi@aut.ac.ir**

# XML

- **Structure of XML Data**
- XML Document Schema
- Querying and Transformation
- Application Program Interfaces to XML
- Storage of XML Data
- XML Applications

# Introduction

- XML:  Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML

# Introduction

- Markup: anything in a document that is not intended to be part of the printed output.

- Such notes convey extra information about the text.
  - For example, a typeset in a magazine may want to make notes about how the typesetting should be done. These notes should be distinguished from the actual content, so that they do not end up printed in the magazine.

- In electronic document processing, a markup language is a formal description of what part of the document is content, what part is markup, and what the markup means.

# Introduction

- Documents have tags giving extra information about sections of the document
  - E.g.

    <title> XML </title>

    <slide> Introduction …</slide>

- **Extensible**, unlike HTML
  - Users can add new tags, and *separately* specify how the tag should be handled for display

# Introduction

- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
  - Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
  - E.g.

```
<university>
      <department>
         <dept_name> Comp. Sci. </dept_name>
         <building> Taylor </building>
         <budget> 100000 </budget>
     </department>
     <course>
          <course_id> CS-101 </course_id>
          <title> Intro. to Computer Science </title>
          <dept_name> Comp. Sci </dept_name>
          <credits> 4 </credits>
      </course>
</university>
```

# Motivation

- Data interchange is critical in today’s networked world
  - Examples:
    - 4 Banking:  funds transfer
    - 4 Order processing (especially inter-company orders)
    - 4 Scientific data
      - Chemistry:  ChemML, …
      - Genetics:    BSML (Bio-Sequence Markup Language), …
  - Paper flow of information between organizations is being replaced by electronic flow of information

# Example

- Purchase orders are typically generated by one organization and sent to another.
- Traditionally they were printed on paper by the purchaser and sent to the supplier; the data would be manually re-entered into a computer system by the supplier.
- This slow process can be greatly sped up by sending the information electronically between the purchaser and supplier.
- XML provides a standard way of tagging the data.
- The two organizations must of course agree on what tags appear in the purchase order, and what they mean

# Example of XML Data

```
  <purchase_order>
       <identifier> P-101 </identifier>
       <purchaser>  …. </purchaser>
       <itemlist>
          <item>
              <identifier> RS1 </identifier>
              <description> Atom powered rocket sled </description>
              <quantity> 2 </quantity>
              <price> 199.95 </price>
          </item>
          <item>
              <identifier> SG2 </identifier>
              <description> Superb glue </description>
              <quantity> 1 </quantity>
              <unit-of-measure> liter </unit-of-measure>
              <price> 29.95 </price>
          </item>
        </itemlist>
       <total cost> 429.85 </total cost>
       <payment terms> Cash-on-delivery </payment terms>
</purchase_order>
```

# Motivation

- Each application area has its own set of standards for representing information

- XML has become the basis for all new generation data interchange formats

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
  - Similar in concept to email headers
  - Does not allow for nested structures, no standard “type” language
  - Tied too closely to low level document structure (lines, spaces, etc)

# Motivation

- Each XML based standard defines what are valid elements, using
  - XML type specification languages to specify the syntax
    - DTD (Document Type Descriptors)
    - XML Schema
  - Plus textual descriptions of the semantics
- XML allows new tags to be defined as required
  - However, this may be constrained by DTDs
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

# Comparison with Relational Data

- Inefficient: tags, which in effect represent schema information, are repeated

- Better than relational tuples as a data-exchange format
  - Unlike relational tuples, XML data is self-documenting due to presence of tags
  - Non-rigid format: tags can be added or ignored
    - 4 E.g., <unit-of-measure>
  - Allows nested structures
  - Wide acceptance, not only in database systems, but also in browsers, tools, and applications

# Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with <*tagname*> and ending with matching </*tagname*>
- Every document must have a single top-level root element
  - That encompasses all other elements in the document
  - E.g., <university> ... </university>

- Elements must be properly nested
  - Proper nesting
    - 4 <course> … <title> …. </title> </course>
  - Improper nesting
    - 4 <course> … <title> …. </course> </title>
  - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.

# Motivation for Nesting

- Each purchase order has a purchaser and a list of items as two of its nested structures.
  - Each item in turn has an item identifier, description and a price nested within it.
  - The purchaser has a name and address nested within it.

- Such information are normally split into multiple relations in a relational schema:
  - Item information
  - Purchaser information
  - Purchase orders
  - The relationship between purchase orders, purchasers, and items

# Motivation for Nesting

- The relational representation helps to avoid redundancy.
  - E.g., item descriptions would be stored only once for each item identifier in a normalized relational schema.

- In the XML purchase order, however, the descriptions may be repeated in multiple purchase orders that order the same item.

- Gathering all information related to a purchase order into a single nested structure, even at the cost of redundancy, is attractive when information has to be exchanged with external parties.

# Structure of XML Data

- Mixture of text with sub-elements is legal in XML.
  - Example:

```
<course>
   This course is being offered for the first time in 2009.
   <course id> BIO-399 </course id>
   <title> Computational Biology </title>
   <dept name> Biology </dept name>
   <credits> 3 </credits>
</course>
```

  - Useful for document processing rather than data processing which deals with structured data

# Attributes

- Elements can have **attributes**

```
<course course_id= “CS-101”>
     <title> Intro. to Computer Science</title>
     <dept name> Comp. Sci. </dept name>
     <credits> 4 </credits>
 </course>
```

- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

```
<course  course_id = “CS-101”  credits=“4”>
```

# Attributes vs. Subelements

- Distinction between subelement and attribute
  - In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents
  - In the context of data representation, the difference is unclear and may be confusing
    - Same information can be represented in two ways
      - <course course_id= “CS-101”> … </course>

      -
        ```
        <course>
            <course_id>CS-101</course_id> …
         </course>
        ```
  - Suggestion: use attributes for identifiers of elements, and use subelements for other contents

# Namespaces

- XML data has to be exchanged between organizations
- Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- Specifying a unique string as an element name avoids confusion
- Better solution: use unique-name:element-name
- Avoid using long unique names all over document by using XML Namespaces

```
<university xmlns:yale="http://www.yale.edu">
     …
      <yale:course>
          <yale:course_id> CS-101 </yale:course_id>
          <yale:title> Intro. to Computer Science</yale:title>
          <yale:dept_name> Comp. Sci. </yale:dept_name>
          <yale:credits> 4 </yale:credits>
      </yale:course>
     …
</university>
```

# More on XML Syntax

- Elements without subelements or text content can be abbreviated by ending the start tag with a /> and deleting the end tag
  - <course course_id=“CS-101” Title=“Intro. To Computer Science” dept_name = “Comp. Sci.” credits=“4” />
- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - <![CDATA[<course> … </course>]]>

Here, <course> and </course> are treated as just strings

CDATA stands for “character data”

# XML

- Structure of XML Data
- **XML Document Schema**
- Querying and Transformation
- Application Program Interfaces to XML
- Storage of XML Data
- XML Applications

# XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
- However, schemas are very important for XML data exchange
  - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema
  - **Document Type Definition (DTD)**
    - 4 Widely used
  - **XML Schema**
    - 4 Newer, increasing use

# Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD

- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.

- DTD does not constrain data types like integer ...
  - All values represented as strings in XML

- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes) >

# Element Specification in DTD

- Subelements can be specified as
  - names of elements, or
  - #PCDATA (parsed character data), i.e., character strings
  - EMPTY (no subelements) or ANY (anything can be a subelement)

- Example

```
<! ELEMENT department (dept_name  building, budget)>
<! ELEMENT dept_name (#PCDATA)>
<! ELEMENT building(#PCDATA)>
<! ELEMENT budget (#PCDATA)>
```

# University DTD

```
<!DOCTYPE  university [
    <!ELEMENT university ( (department|course|instructor|teaches)+)>
    <!ELEMENT department ( dept name, building, budget)>
    <!ELEMENT course ( course id, title, dept name, credits)>
    <!ELEMENT instructor (IID, name, dept name, salary)>
    <!ELEMENT teaches (IID, course id)>
    <!ELEMENT dept name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
    <!ELEMENT IID( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT salary( #PCDATA )>
]>
```

# Element Specification in DTD

- Subelement specification may have regular expressions

  <!ELEMENT university ( ( department | course | instructor | teaches )+)>

  4 Notation:
  - “|” - or
  - “+” - 1 or more occurrences
  - “*” - 0 or more occurrences
  - “?” - 0 or 1 occurrence

# Attribute Specification in DTD

- Attribute specification : for each attribute
  - Name
  - Type of attribute
    - CDATA
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
  - Whether
    - mandatory (#REQUIRED)
      - has a default value (value),
    - or neither (#IMPLIED)
- Examples
  - <!ATTLIST course course_id CDATA #REQUIRED>, or
  - <!ATTLIST course

```
    course_id     ID         #REQUIRED
    dept_name  IDREF   #REQUIRED
    instructors    IDREFS #IMPLIED   >
```

# IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
    - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of another element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document

# University DTD with Attributes

```
<!DOCTYPE university-3 [
    <!ELEMENT university ( (department|course|instructor)+)>
    <!ELEMENT department ( building, budget )>
    <!ATTLIST department
          dept_name ID #REQUIRED >
    <!ELEMENT course (title, credits )>
    <!ATTLIST course
          course_id ID #REQUIRED
          dept_name IDREF #REQUIRED
          instructors IDREFS #IMPLIED >
    <!ELEMENT instructor ( name, salary )>
    <!ATTLIST instructor
          IID ID #REQUIRED
          dept_name IDREF #REQUIRED >
    . . .
]>
```

# XML data with ID and IDREF attributes

```
<university-3>
    <department dept name=“Comp. Sci.”>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept name=“Biology”>
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course id=“CS-101” dept name=“Comp. Sci” instructors=“10101 83821”>
         <title> Intro. to Computer Science </title>
         <credits> 4 </credits>
    </course>
    ….
    <instructor IID=“10101” dept name=“Comp. Sci.”>
         <name> Srinivasan </name>
         <salary> 65000 </salary>
    </instructor>
    ….
</university-3>
```

# Limitations of DTDs

- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.

- Difficult to specify unordered sets of subelements
  - (A | B)* allows specification of an unordered set
    - 4 Cannot ensure the number of times each of A and B occurs

- IDs and IDREFs are untyped
  - The *instructors* attribute of an course may contain a reference to a department or another course, which is meaningless
    - 4 *instructors* attribute should ideally be constrained to refer to instructor elements

# XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs.

- Supports typing
  - Built-in types
    - E.g., string, integer, decimal, and boolean
    - Also, constraints on min/max values
  - User-defined types
    - Simple types with added restrictions
    - Complex types constructed using constructors
  - Many more features, including
    - uniqueness and foreign key constraints, inheritance

# XML Schema

- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
- XML Scheme is integrated with namespaces; E.g., “xs:”
- BUT:  XML Schema is significantly more complicated than DTDs.

# XML Schema Version of Univ. DTD

```
<xs:schema xmlns:xs=“http://www.w3.org/2001/XMLSchema”>
<xs:element name=“university” type=“universityType” />
<xs:element name=“department”>
     <xs:complexType>
          <xs:sequence>
              <xs:element name=“dept name” type=“xs:string”/>
              <xs:element name=“building” type=“xs:string”/>
              <xs:element name=“budget” type=“xs:decimal”/>
          </xs:sequence>
     </xs:complexType>
</xs:element>
….
<xs:element name=“instructor”>
    <xs:complexType>
        <xs:sequence>
            <xs:element name=“IID” type=“xs:string”/>
            <xs:element name=“name” type=“xs:string”/>
            <xs:element name=“dept name” type=“xs:string”/>
            <xs:element name=“salary” type=“xs:decimal”/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

… Contd.

# XML Schema Version of Univ. DTD (Cont.)

```
….
<xs:complexType name=“UniversityType”>
    <xs:sequence>
        <xs:element ref=“department” minOccurs=“0” maxOccurs=“unbounded”/>
        <xs:element ref=“course” minOccurs=“0” maxOccurs=“unbounded”/>
        <xs:element ref=“instructor” minOccurs=“0” maxOccurs=“unbounded”/>
        <xs:element ref=“teaches” minOccurs=“0” maxOccurs=“unbounded”/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

- Choice of “xs:” was ours -- any other namespace prefix could be chosen
- Element “university” has type “universityType”, which is defined separately
    - xs:complexType is used later to create the named complex type “UniversityType”

# More features of XML Schema

- Attributes specified by xs:attribute tag:
  - <xs:attribute name = “dept_name”/>
  - adding the attribute use = “required” means value must be specified

# More features of XML Schema

- Key constraint: “department names form a key for department elements under the root university element:

```
<xs:key name = “deptKey”>
        <xs:selector xpath = “/university/department”/>
        <xs:field xpath = “dept_name”/>
<\xs:key>
```

- Foreign key constraint from course to department:

```
<xs:keyref name = “courseDeptFKey” refer=“deptKey”>
        <xs:selector xpath = “/university/course”/>
        <xs:field xpath = “dept_name”/>
<\xs:keyref>
```

# XML Schema vs DTD

- Constraining elements texts to specific types; e.g., numeric types or complex types
- User-defined types
- Uniqueness and foreign-key constraints.
- Integrating with namespaces to allow different parts of a document to conform to different schemas.
- Restricting types to create specialized types, for instance by specifying minimum and maximum values.
- Complex types to be extended by using a form of inheritance.

# XML

- Structure of XML Data
- XML Document Schema
- **Querying and Transformation**
- Application Program Interfaces to XML
- Storage of XML Data
- XML Applications

# Querying and Transforming XML Data

- Main applications
  - Extract information from large bodies of XML data
  - Converting data between different representations in XML

- Main requirement: tools for
  - Translation of information from one XML schema to another
  - Querying on XML data

- Above two are closely related, and handled by the same tools

# Querying and Transforming XML Data

- Standard XML querying/translation languages
  - XPath
    - Simple language consisting of path expressions
    - a building block for XQuery
  - XQuery
    - An XML query language with a rich set of features
    - It is modeled after SQL but is significantly different, since it has to deal with nested XML data.
    - XQuery also incorporates XPath expressions.
  - XSLT
    - Simple language designed for translation from XML to XML and XML to HTML
    - It is used primarily in document-formatting applications, rather in data management applications

# Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data

- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
  - Element nodes have child nodes, which can be attributes or subelements
  - Text in an element is modeled as a text node child of the element
  - Children of a node are ordered according to their order in the XML document
  - Element and attribute nodes (except for the root node) have a single parent, which is an element node
  - The root node has a single child, which is the root element of the document

# XPath

- XPath is used to address (select) parts of documents using **path expressions**

- A path expression is a sequence of steps separated by “/”
  - Think of file names in a directory hierarchy

- Result of path expression: set of values that along with their containing elements/attributes match the specified path
  - E.g. /university-3/instructor/name returns

    <name>Srinivasan</name>
    <name>Brandt</name>

  - E.g. /university-3/instructor/name/text( ) returns

    Srinivasan
    Brandt

- The nodes returned by each step appear in the same order as their appearance in the document.

# XPath

- The initial “/” denotes root of the document (above the top-level tag)

- Path expressions are evaluated left to right
  - Each step operates on the set of instances produced by the previous step

- Attribute values are accessed, using the “@” symbol.
  - E.g. /university-3/course/@course_id returns

    a set of all values of course_id attributes of course elements

- IDREF attributes are not dereferenced automatically (more on this later)

# XPath

- Selection predicates may follow any step in a path, in [ ]
  - E.g. /university-3/course[credits] returns
    - 4 course elements containing a credits subelement
  - E.g. /university-3/course[credits >= 4] returns
    - 4 course elements with a credits value greater than or equal to 4,
  - E.g. /university-3/course[credits >= 4]/@course_id returns
    - 4 returns the course identifiers of courses with credits >= 4

# Functions in XPath

- XPath provides several functions
- The function count() at the end of a path counts the number of elements in the set generated by the path
  - E.g. /university-2/instructor[count(./teaches/course)> 2] returns
    - instructors teaching more than 2 courses (on university-2 schema)
- Also function for testing position (1, 2, ..) of node w.r.t. siblings
- Boolean connectives and and or and function not() can be used in predicates

# Functions in XPath

- The function id("k") returns the node (if any) with an attribute of type ID and value "k".

- IDREFs can be referenced using function id()
  - id() can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks

  - E.g. /university-3/course/id(@dept_name) returns
    - 4 all department elements referred to from the dept_name attribute of course elements.

  - E.g. /university-3/course/id(@instructors) returns
    - 4 all instructor elements referred to from the instructor attribute of course elements.

# More XPath Features

- Operator “|” used to implement union
  - E.g. /university-3/course[@dept name=“Comp. Sci”] | /university-3/course[@dept name=“Biology”]
    - 4 Gives union of Comp. Sci. and Biology courses
    - 4 However, “|” cannot be nested inside other operators.
    - 4 The nodes in the union are returned in the order in which they appear in the document

# More XPath Features

- doc(name) returns the root of a named document

- Thus, a path expression can be applied on a specified document, instead of being applied on the current default document.
    - E.g. doc("university.xml")/university/department returns
        - 4 all departments at the university

- The function collection(name) is similar to doc, but returns a collection of documents identified by name

# More XPath Features

- “//” can be used to skip multiple levels of nodes
  - E.g. /university-3//name
    - finds any name element *anywhere* under the /university-3 element, regardless of the element in which it is contained.

- A step in the path can go to parents, siblings, ancestors and descendants of the nodes generated by the previous step, not just to the children
  - “//”, described above, is a short form for specifying “all descendants”
  - “..” specifies the parent.

# XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
- XQuery queries are modeled after SQL queries, but differ significantly from SQL .

# XQuery

- XQuery is organized into five sections

  **for … let … where … order by …return …**

- XQuery syntax

  **for** ⇔ SQL **from**

  **let** allows temporary variables, and has no equivalent in SQL

  **where** ⇔ SQL **where**

  **order by** ⇔ SQL **order by**

  **return** ⇔ SQL **select** (allows the construction of results in XML)

- It is referred to as “ FLWOR ” (pronounced “flower”) expression
- A FLWOR query need not contain all the clauses

# FLWOR Syntax in XQuery

- Simple FLWOR expression in XQuery
  - find all courses with credits > 3, with each result enclosed in an <course_id> .. </course_id> tag

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

# FLWOR Syntax in XQuery

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- The where clause, like the SQL where clause, performs additional tests on the joined tuples from the for clause
- Since the for clause uses XPath expressions, selections may occur within the XPath expression.

```
for $x in /university-3/course[credits > 3]
```

# FLWOR Syntax in XQuery

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

- The let clause simply allows the results of XPath expressions to be assigned to variable names for simplicity of representation
- Let clause not really needed in this query
- In fact, since this query is simple, we can easily do away with the let clause, and the variable $courseId in the return clause could be replaced with $x/@course id.

```
for $x in /university-3/course[credits > 3]
return <course_id> { $x/@course_id } </course_id>
```

# FLWOR Syntax in XQuery

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

- Items in the return clause are XML text unless enclosed in {}, in which case they are evaluated as expressions

- The query can be modified to return an element with tag course, with the course identifier as an attribute, by replacing the return clause with the following:

```
return <course course id="{$x/@course id}" />
```

# FLWOR Syntax in XQuery

```
for  $x in /university-3/course
let   $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

- XQuery also supports constructing elements using the element and attribute constructors.
  - E.g.,

```
return element course {
        attribute course_id {$x/@course_id},
        attribute dept_name {$x/dept_name},
        element title {$x/title},
        element credits {$x/credits}
}
```

returns course elements with course_id and dept_name as attributes and title and credits as subelements.

# Joins

- Joins are specified in a manner very similar to SQL

```
 for $c in /university/course,
      $i in /university/instructor,
      $t in /university/teaches
where $c/course_id= $t/course id and $t/IID = $i/IID
return <course_instructor> { $c $i } </course_instructor>
```

- The same query can be expressed with the selections specified as XPath selections:

```
for $c in /university/course,
     $i in /university/instructor,
     $t in /university/teaches[ $c/course_id= $t/course_id
                                                  and $t/IID = $i/IID]
return <course_instructor> { $c $i } </course_instructor>
```

# Nested Queries

- XQuery FLWOR expressions can be nested in the return clause, in order to generate element nestings that do not appear in the source document

```
 <university-1>
{     for $d in /university/department
       return <department>
                     { $d/* }
                     { for $c in /university/course[dept name = $d/dept name]
                       return $c }
                 </department>
}
{      for $i in /university/instructor
       return  <instructor>
                      { $i/* }
                      { for $c in /university/teaches[IID = $i/IID]
                        return $c/course id }
                  </instructor>
}
</university-1>
```

- $d/* denotes all the children of the node to which $d is bound, without the enclosing top-level tag

# Nested Queries (Example Input)

```
<university>
        <department>
            <dept_name> Comp. Sci. </dept_name>
            <building> Taylor </building>
            <budget> 100000 </budget>
        </department>
       <department>
            <dept name> Biology </dept name>
            <building> Watson </building>
            <budget> 90000 </budget>
       </department>
      <course>
          <course_id> CS-101 </course_id>
          <title> Intro. to Computer Science </title>
          <dept_name> Comp. Sci </dept_name>
          <credits> 4 </credits>
      </course>
       <course>
            <course id> BIO-301 </course id>
            <title> Genetics </title>
            <dept name> Biology </dept name>
            <credits> 4 </credits>
       </course>
```

# Nested Queries (Example Input)

```
 ...
<instructor>
       <IID> 10101 </IID>
       <name> Srinivasan </name>
       <dept name> Comp. Sci. </dept name>
 </instructor>
 <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept name> Biology </dept name>
</instructor>
<teaches>
       <IID> 10101 </IID>
       <course id> CS-101 </course id>
</teaches>
<teaches>
       <IID> 76766 </IID>
       <course id> BIO-301 </course id>
 </teaches>
</university>
```

# Nested Queries (Example Output)

```
<university-1>
    <department>
        <dept name> Comp. Sci. </dept name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course id> CS-101 </course id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
    </department>
    <department>
        <dept name> Biology </dept name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course id> BIO-301 </course id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
```

# Nested Queries (Example Output)

```
...
        <instructor>
                <IID> 10101 </IID>
                <name> Srinivasan </name>
                <dept name> Comp. Sci. </dept name>
                <course id> CS-101 </coursr id>
        </instructor>
        <instructor>
                <IID> 76766 </IID>
                <name> Crick </name>
                <dept name> Biology </dept name>
                <course id> BIO-301 </coursr id>
        </instructor>
</university-1>
```

# Grouping and Aggregation

- XQuery provides a variety of aggregate functions such as sum() and count() that can be applied on sequences of elements or values.

- To avoid namespace conflicts, functions are associated with a namespace:

  http://www.w3.org/2005/xpath-functions

- which has a default namespace prefix of *fn*. Thus, these functions can be referred to unambiguously as fn:sum or fn:count.

# Grouping and Aggregation

- Nested queries are used for grouping

```
for $d in /university/department
return
       <department-total-salary>
            <dept_name> { $d/dept name } </dept_name>
             <total_salary> { fn:sum(
                  for $i in /university/instructor[dept_name =$d/dept_name]
                  return $i/salary
                ) }
             </total_salary>
        </department-total-salary>
```

# Sorting in XQuery

- The **order by** clause can be used at the end of any expression. E.g. to return instructors sorted by name

```
for $i in /university/instructor
order by $i/name
return <instructor> { $i/* } </instructor>
```

- Use **order by** $i/name **descending** to sort in descending order

# Sorting in XQuery

- Can sort at multiple levels of nesting (sort departments  by dept_name, and by courses sorted to course_id within each department)

```
<university-1> {
  for $d in /university/department
  order by $d/dept name
  return
       <department>
           { $d/* }
           { for $c in /university/course[dept name = $d/dept name]
             order by $c/course id
             return <course> { $c/* } </course> }
       </department>
} </university-1>
```

# XSLT

- A **stylesheet** stores formatting options for a document, usually separately from document
  - E.g. an HTML style sheet may specify font colors and sizes for headings, etc.

- The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML

- XSLT is a general-purpose transformation language
  - Can translate XML to XML, and XML to HTML

- XSLT transformations are expressed using rules called **templates**
  - Templates combine selection using XPath with construction of results

# XML

- Structure of XML Data
- XML Document Schema
- Querying and Transformation
- **Application Program Interfaces to XML**
- Storage of XML Data
- XML Applications

# Application Program Interface

- One of the standard APIs for manipulating XML is based on the document object model ( DOM ).

- DOM treats XML content as a tree, with each element represented by a node, called a DOMNode.

- Programs may access parts of the document in a navigational fashion, beginning with the root.

- DOM libraries are available for most common programming languages and are even present in Web browsers, where they may be used to manipulate the document displayed to the user.

- It also provides functions for updating DOM tree

# Document Object Model

- The Java DOM API provides an interface called Node, and interfaces Element and Attribute, which inherit from the Node interface.

- The Node interface provides methods for navigating the DOM tree, starting with the root node. E.g.,
  - getParentNode( )
  - getFirstChild( )
  - getNextSibling( )

- Subelements of an element can be accessed by name:
  - getElementsByTagName( ) returns a list of all child elements with a specified tag name
  - Individual members of the list can be accessed by the method item(i), which returns the ith element in the list.

# Document Object Model

- Attribute values of an element can be accessed by name, using the method getAttribute( ).

- The text value of an element is modeled as a Text node, which is a child of the element node
  - an element node with no subelements has only one such child node.
  - The method getData() on the Text node returns the text contents.

# XML

- Structure of XML Data
- XML Document Schema
- Querying and Transformation
- Application Program Interfaces to XML
- **Storage of XML Data**
- XML Applications

# Storage of XML Data

- XML data can be stored in
  - Non-relational data stores
    - Flat files
      - Natural for storing XML
      - But has all problems discussed in Chapter 1
        - Data isolation
        - Atomicity
        - Concurrent access
        - Security
    - XML database
      - Database built specifically for storing XML data, supporting DOM model and declarative querying
      - Currently no commercial-grade systems

# Storage of XML Data

- XML data can be stored in
  - Relational databases
    - 4 Data must be translated into relational form
    - 4 Advantage:  mature database systems
    - 4 Disadvantages: overhead of translating data and queries

# Storage of XML in Relational Databases

- Approaches:
  - String Representation
  - Tree Representation
  - Map to relations

# String Representation

- Store each top level element as a string field of a tuple in a relational database
  - Use a single relation to store all elements
    - 4 Easy to use, but no knowledge about the schema of the stored elements
    - 4 Not possible to query the data directly
  - Use a separate relation for each top-level element type
    - 4 E.g. department, course, instructor, and teaches
      - – Each with a string-valued attribute to store the element

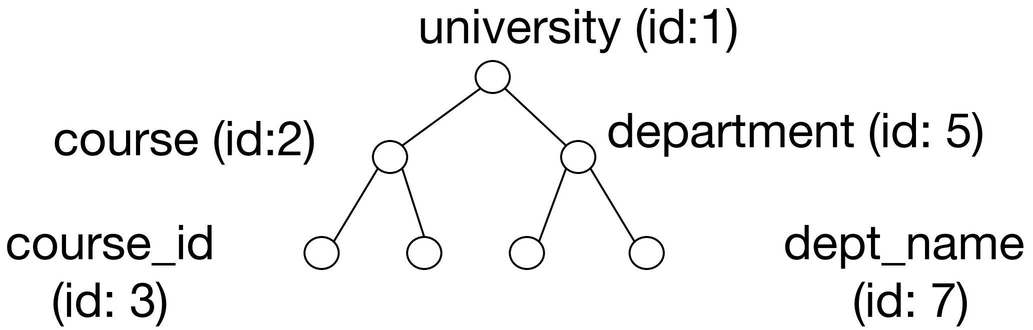
- Indexing:
  - Store values of subelements/attributes to be indexed as extra fields of the relation, and build indices on these fields
    - 4 E.g. dept_name, course id, or name

# String Representation

- Benefits:
  - Can store any XML data even without DTD
  - As long as there are many top-level elements in a document, strings are small compared to full document
    - 4 Allows fast access to individual elements.
- Drawback**:** Need to parse strings to access values inside the elements
  - Parsing is slow.

# Tree Representation

- **Tree representation:** model XML data as tree and store using relations
  *nodes(id, parent_id, type, label, value)*



- Each element/attribute is given a unique identifier
- Type indicates element/attribute
- Label specifies the tag name of the element/name of attribute
- Value is the text value of the element/attribute
- Can add an extra attribute *position* to record ordering of children

# Tree Representation

- Benefit:
  - Can store any XML data, even without DTD

- Drawbacks:
  - Data is broken up into too many pieces, increasing space overheads
  - Even simple queries require a large number of joins, which can be slow

# Mapping XML Data to Relations

- Relation created for each element type whose schema is known:
  - An id attribute to store a unique id for each element
  - A relation attribute corresponding to each element attribute
  - A parent_id attribute to keep track of parent element
    - As in the tree representation
    - Position information ($i^{th}$ child) can be store too
- All subelements that occur only once can become relation attributes
  - For text-valued subelements, store the text as attribute value
  - For complex subelements, can store the id of the subelement
- Subelements that can occur multiple times represented in a separate table
  - Similar to handling of multivalued attributes when converting ER diagrams to tables

# Storing XML Data in Relational Systems

- Applying above ideas to department elements in university-1 schema, with nested course elements, we get

  *department*(*id*, *dept_name*, *building*, *budget*)
  *course*(*parent_id*, *course_id*, *dept_name*, *title*, *credits*)

# Publishing and Shredding XML Data

- **Publishing**: process of converting relational data to an XML format for export to other applications.

- **Shredding**: process of converting back an XML document into a set of tuples to be inserted into one or more relations

- *XML-enabled* database systems support automated publishing and shredding
- Otherwise, application code can perform the publishing and shredding operations

# Publishing and Shredding XML Data

- An XML-enabled database supports automatic publishing using map to relation mechanism

- Approaches:
  - Simple
    - Creates an XML element for every row of a table
    - Makes each column in that row a subelement of the XML element
  - Complex
    - Allows nested structures to be created
    - Extensions of SQL with nested queries in the select clause have been developed to allow easy creation of nested XML output

# SQL/XML

- New standard SQL extension that allows creation of nested XML output
  - Each output tuple is mapped to an XML element *row*
  - Each relation attribute mapped to an XML element of the same name

# SQL/XML

```
<university>
     <department>
          <row>
               <dept name> Comp. Sci. </dept name>
               <building> Taylor </building>
               <budget> 100000 </budget>
          </row>
          <row>
               <dept name> Biology </dept name>
               <building> Watson </building>
               <budget> 90000 </budget>
          </row>
     </department>
     <course>
          <row>
               <course id> CS-101 </course id>
               <title> Intro. to Computer Science </title>
               <dept name> Comp. Sci </dept name>
               <credits> 4 </credits>
          </row>
 </university>
```

# SQL Extensions

- **xmlelement** creates XML elements
- **xmlattributes** creates attributes

```
select xmlelement (name "course",
        xmlattributes (course id as course id, dept name as dept name),
        xmlelement (name "title", title),
        xmlelement (name "credits", credits))
from course
```

# SQL Extensions

- Xmlagg creates a forest of XML elements

- Creating an element for each department with a course, containing as subelements all the courses in that department.
- Since the query has a clause group by dept_name, the aggregate function is applied on all courses in each department, creating a sequence of course id elements.

```
select xmlelement (name “department”, dept_name,
            xmlagg (xmlforest(course_id)
                        order by course_id))
from course
group by dept_name
```

# XML

- Structure of XML Data
- XML Document Schema
- Querying and Transformation
- Application Program Interfaces to XML
- Storage of XML Data
- **XML Applications**

# XML Applications

- Storing data with complex structures
  - Applications that need to store data that are structured, but are not easily modeled as relations
    - 4 E.g., user preferences stored by a Web browser
  - Editable document representation as part of office application packages for storing documents
    - 4 E.g., Open Office: Open Document Format (ODF)
    - 4 E.g., Microsoft Office: Office Open XML (OOXML)

# XML Applications

- Standardized data exchange formats
  - E.g., ChemML: a standard for representing/exchanging information about chemicals
    - Molecular structure
    - Boiling and melting points
    - Calorific values
    - Solubility in various solvents
  - E.g., required information for business-to-business (B2B) market
    - Product descriptions
    - Price information
    - Product inventories
    - Quotes for a proposed sale

# XML Applications

- Standard for data exchange for Web services
  - Organizations are not willing to allow direct access to their database using SQL, but normally provide limited forms of information through predefined interfaces.
    - When the information is to be used directly by a human, organizations provide Web-based forms, where users can input values and get back desired information in HTML form.
    - When such information needs to be accessed by software programs rather than by end users, the input and output are normally provided in XML form
      - The HTTP protocol is used to communicate the input and output information
      - The Simple Object Access Protocol (SOAP) defines a standard for invoking procedures, using XML for representing the procedure input and output

# XML Applications

- Data mediation
  - Common data representation format to bridge different systems
    - 4 E.g., a consumer with a variety of bank/credit-card accounts held at different institutions
    - 4 Once the basic tools are available to extract information from each source, a mediator application is used to combine the extracted information under a single schema.
    - 4 This may require further transformation of the XML data from each site due to different structures and different names for the same information (for instance, acct_number and account_id), or may even use the same name for different information
  - XML query languages such as XSLT and XQuery play an important role in the task of transformation between different XML representations.

# Questions?