SQL and the Web

XML/JSON and Semi-Structured Data

Databases No Longer Stand Alone

- Who writes SQL queries at the Oracle prompt?

- We'll discuss two scenarios where a DB is part of a larger ecosystem:
  - Within a standalone application

    Focus for now is on SQL interface…

  - As part of a Web application

Why XML (and JSON)?

XML is the confluence of several factors:

- The Web needed a more declarative format for data – human and machine readable
- Documents needed a mechanism for extended tags
- Database people needed a more flexible interchange format
- It's parsable even if we don't know what it means!

Original expectation:

- The whole web would go to XML instead of HTML

Today's reality:

- Not so… But XML (and JSON) are used all over "under the covers"

Why DB People Care about XML

Can get data from all sorts of sources

- Allows us to touch data we don't own!
- Documents can be data too!
- This was actually a huge change in the DB community

Interesting relationships with DB techniques

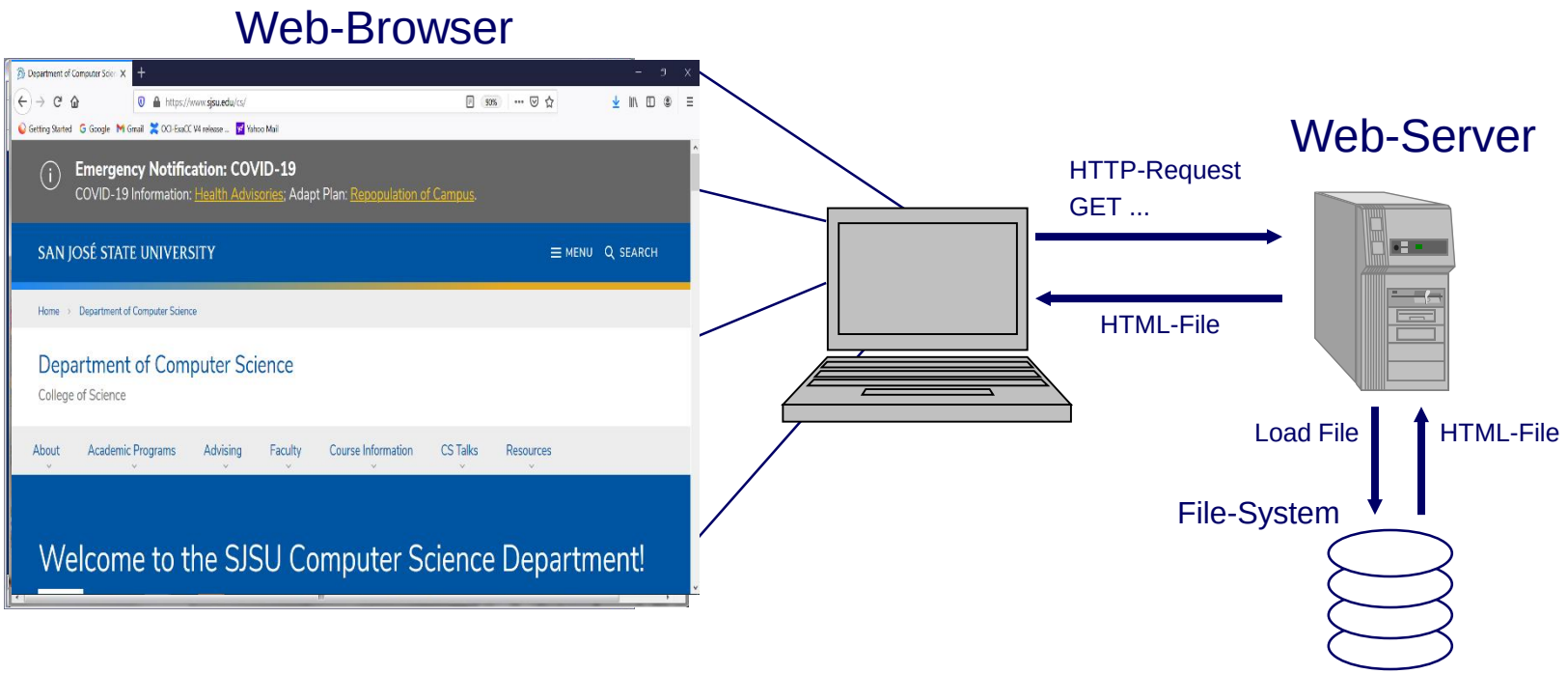
- Useful to do relational-style operations
- Leverages ideas from object-oriented, semistructured data

Blends schema and data into one format

- Unlike relational model, where we need schema first
- … But too little schema can be a drawback, too!

Database-Backed Web Sites

- We all know traditional static HTML web sites:



How about a Modern Web Site?

- We know that modern Web sites are now a combination of pieces:
  - Database and/or key-value-store backend
    - Possibly a caching layer
  - Server-side dynamic code
  - Client-side dynamic code

A Big (and Frustrating) Issue

- When I pass data from the server to the client, and vice versa, how do I do it?
  - Server-side language and client-side language (JavaScript) are often NOT the same!
  - So we need to **marshall** objects from one side to the other, and convert them
    - ***And we need type/class definitions on both sides!***
  - Typically the data is passed in JSON form – JavaScript Object Notation

JSON

JSON example

- "JSON" stands for "JavaScript Object Notation"
  - Despite the name, JSON is a (mostly) language-independent way of specifying objects as name-value pairs
- Example (http://secretgeek.net/json_3mins.asp):
  -
```
{"skillz": {
    "web":[
        { "name": "html",
          "years": 5
        },
        { "name": "css",
          "years": 3
        }]
    "database":[
        { "name": "sql",
          "years": 7
        }]
}}
```

JSON syntax, I

- An *object* is an unordered set of name/value pairs
  - The pairs are enclosed within braces, { }
  - There is a colon between the name and the value
  - Pairs are separated by commas
  - Example: { "name": "html", "years": 5 }
- An *array* is an ordered collection of values
  - The values are enclosed within brackets, [ ]
  - Values are separated by commas
  - Example: ["html", ”xml", "css"]

JSON syntax, II

- A *value* can be: A string, a number, true, false, null, an object, or an array
  - Values can be nested
- *Strings* are enclosed in double quotes, and can contain the usual assortment of escaped characters
- *Numbers* have the usual C/C++/Java syntax, including exponential (E) notation
  - All numbers are decimal--no octal or hexadecimal
- *Whitespace* can be used between any pair of tokens

Using JSON in JavaScript

- Need a JSON parser or a function, **`stringify()`**, to convert between JavaScript objects and JSON encoded data.
  - http://www.json.org/json2.js

- JSON encoded data ➔ JavaScript object
  - `var myObject = eval('(' + myJSONtext + ')');`
  - `var myObject = JSON.parse(myJSONtext);`

- JavaScript value ➔ JSON encoded data
  - `var myJSONText = JSON.stringify(myObject);`

Using JSON with XmlHttpRequest

- Sending JSON encoded data to the server
  - Use HTTP POST method and send the JSON encoded data in the body of the request

```
// xmlhttp is an XmlHttpRequest object
xmlhttp.setRequestHeader(
  'Content-type',
  'application/x-www-form-urlencoded;charset=UTF-8;'
);
xmlhttp.send('jsondata=' + escape(myJSONText));
```

- Handling JSON encoded data from the server
  - Server should set the content type to "text/plain"
  - In the handler function of **`xmlhttp`** object, read **`xmlhttp.responseText`**

eval

- The JavaScript eval(***string***) method compiles and executes the given string
  - The string can be an expression, a statement, or a sequence of statements
  - Expressions can include variables and object properties
  - eval returns the value of the last expression evaluated
- When applied to JSON, eval returns the described object

Using JSON in Java

```
import org.json.simple.JSONObject;
import org.json.simple.JSONArray;
……
public class MyServlet extends HttpServlet {
public void
doGet (HttpServletRequest request,HttpServletResponse response)
throws ServletException, IOException
{
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String feedURLString = request.getParameter("feedURL");
String script ="";
JSONObject obj = new JSONObject();
JSONArray arry = new JSONArray();
……

}
```

Speeding Up AJAX with JSON

- Both XML and JSON use structured approaches to mark up data.

- More and more web services are supporting JSON
  - e.g.: Yahoo's various search services, travel planners, and highway traffic services

```
function myHandler() {
   if (req.readyState == 4 /*complete*/) {
       var addrField   = document.getElementById('addr');
       var root        = req.responseXML;
       var addrsElem   = root.getElementsByTagName('addresses')[0];
       var firstAddr   = addrsElem.getElementsByTagName('address')[0];
       var addrText    = fistAddr.firstChild;
       var addrValue   = addrText.nodeValue;
       addrField.value = addrValue;
   }
}
```

JavaScript code to handle XML encoded data

```
function myHandler() {
   if (req.readyState == 4 /*complete*/) {
       var addrField = document.getElementById('addr');
       var card = eval('(' + req.responseText + ')');
       addrField.value = card.addresses[0].value;
   }
}
```

JavaScript code to handle JSON encoded data

Both examples try to update the value of a form element named "addr" with the data obtained from an HTTP request.

XML?

Extensible Markup Language

- Designed to describe data

XML and related technologies (DTD, XML Schema, XPath, XQuery, etc.) have been standardized mainly by the

World Wide Web Consortium (W3C)

http://www.w3.org

More resources at http://www.xml.com

Authoring XML Elements

- An XML element is made up of a start tag, an end tag, and data in between.
- Example:

  <director> Matthew Dunn </director>
- Example of another element with the same value:

  <actor> Matthew Dunn </actor>
- XML tags are case-sensitive:

  <CITY> <City> <city>
- XML can abbreviate empty elements, for example:

  <married> </married> can be abbreviated to

  <married/>

Authoring XML Elements (cont'd)

- An attribute is a name-value pair separated by an equal sign (=).
- Example:

  <City ZIP="94608"> Emeryville </City>

- Attributes are used to attach additional, secondary information to an element.

XML Anatomy

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
  <mastersthesis mdate="2002-01-03" key="ms/Brown92">
    <author>Kurt P. Brown</author>
    <title>PRPL: A Database Workload Specification Language</title>
    <year>1992</year>
    <school>Univ. of Wisconsin-Madison</school>
  </mastersthesis>
  <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
    <editor>Paul R. McJones</editor>
    <title>The 1995 SQL Reunion</title>
    <journal>Digital System Research Center Report</journal>
    <volume>SRC1997-018</volume>
    <year>1997</year>
    <ee>db/labs/dec/SRC1997-018.html</ee>
    <ee>http://www.mcjones.org/System_R/SQL_Reunion_95/</ee>
  </article>
```

Processing Instr.

Open-tag

Element

Attribute

Close-tag

Well-Formed XML

A legal XML document – fully parsable by an XML parser

- All open-tags have matching close-tags (unlike so many HTML documents!), or a special:

  <tag/> shortcut for empty tags (equivalent to <tag></tag>

- Attributes (which are unordered, in contrast to elements) only appear once in an element
- There's a single root element
- XML is case-sensitive

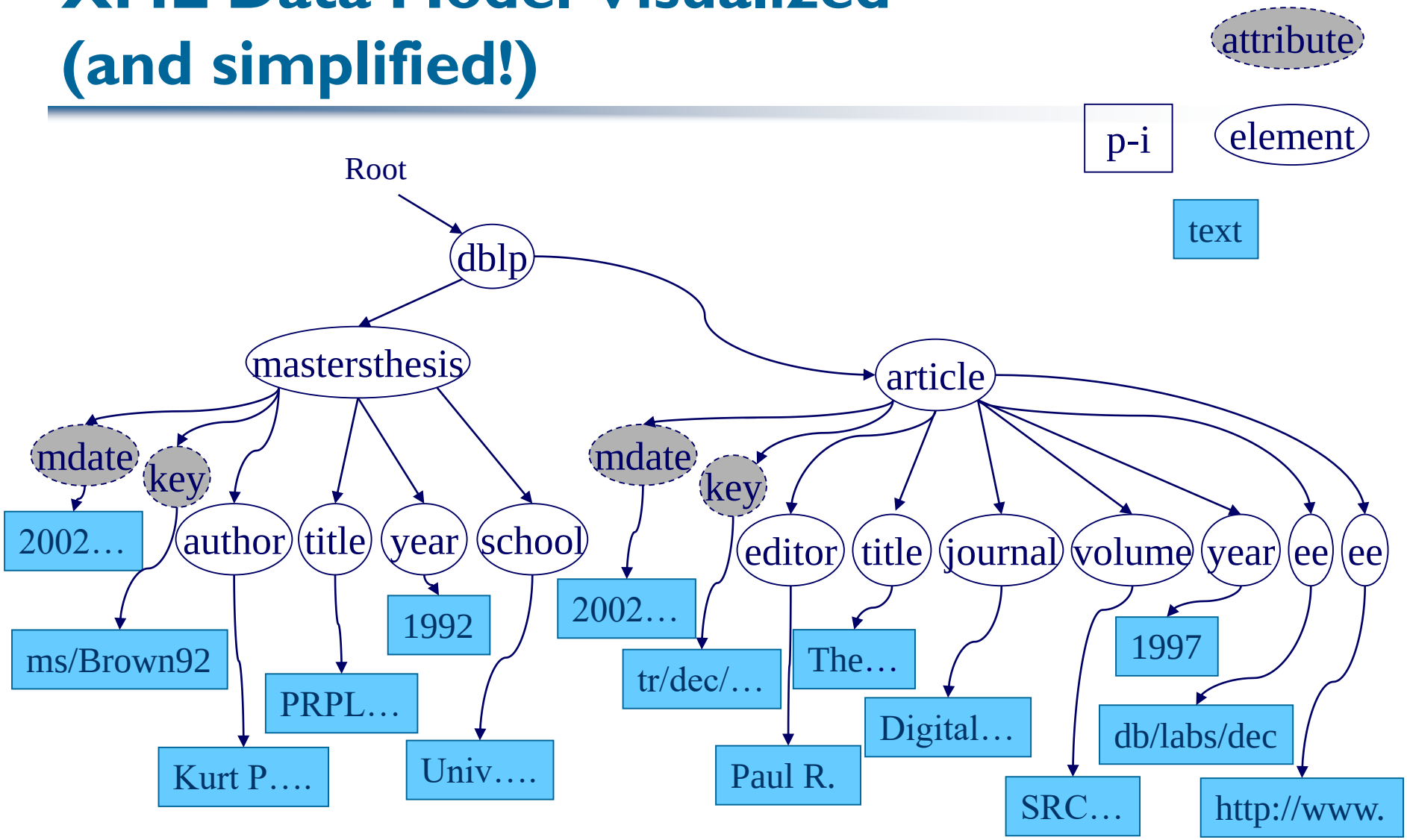
XML as a Data Model

XML "information set" includes 7 types of nodes:

- Document (root)
- Element
- Attribute
- Processing instruction
- Text (content)
- Namespace
- Comment

XML data model includes this, plus order info and a few other things

XML Data Model Visualized (and simplified!)



XML Easily Encodes Relations

Student-course-grade

sid	serno	exp-grade
1	570103	B
23	550103	A

XML Easily Encodes Relations

```
<student-course-grade>
    <tuple>
        <sid>1</sid>
        <serno>570103</serno>
        <exp-grade>B</exp-grade>
    </tuple>
    <tuple>
        <sid>23</sid>
        <serno>550103</serno>
        <exp-grade>A</exp-grade>
    </tuple>
</student-course-grade>
```

But XML is More Flexible…

```
<parents>
  <parent name="Jean" >
    <son>John</son>
    <daughter>Joan</daughter>
    <daughter>Jill</daughter>
  </parent>
  <parent name="Feng">
    <daughter>Felicity</daughter>
  </parent>
…
```

XML Isn't Enough on Its Own

It's too unconstrained for many cases!

- How will we know when we're getting garbage?
- How will we query?
- How will we understand what we got?

We also need:

Some idea of the structure

- Our focus next

Presentation, in some cases – CSS, XSL

- You can read about this separately at http://www.w3.org/Style

XML vs. JSON (in AJAX Application)

- JSON produces slightly smaller documents
- JSON is easier to use in JavaScript
- Parsing JSON encoded data is much faster than parsing XML encoded data

```
<?xml version='1.0' encoding='UTF-8'?>
<card>
   <fullname>Sean Kelly</fullname>
   <org>SK Consulting</org>
   <emailaddrs>
      <address type='work'>kelly@seankelly.biz</address>
      <address type='home' pref='1'>kelly@seankelly.tv</address>
   </emailaddrs>
   <telephones>
      <tel type='work' pref='1'>+1 214 555 1212</tel>
      <tel type='fax'>+1 214 555 1213</tel>
      <tel type='mobile'>+1 214 555 1214</tel>
   </telephones>
   <addresses>
      <address type='work' format='us'>1234 Main St
         Springfield, TX 78080-1216</address>
      <address type='home' format='us'>5678 Main St
         Springfield, TX 78080-1316</address>
   </addresses>
   <urls>
      <address type='work'>http://seankelly.biz/</address>
      <address type='home'>http://seankelly.tv/</address>
   </urls>
</card>
```

Example: An address book data encoded in XML

```
{
   "fullname": "Sean Kelly",
   "org": "SK Consulting",
   "emailaddrs": [
      {"type": "work", "value": "kelly@seankelly.biz"},
      {"type": "home", "pref": 1, "value": "kelly@seankelly.tv"}
   ],
    "telephones": [
      {"type": "work", "pref": 1, "value": "+1 214 555 1212"},
      {"type": "fax", "value": "+1 214 555 1213"},
      {"type": "mobile", "value": "+1 214 555 1214"}
   ],
   "addresses": [
      {"type": "work", "format": "us",
       "value": "1234 Main StnSpringfield, TX 78080-1216"},
      {"type": "home", "format": "us",
       "value": "5678 Main StnSpringfield, TX 78080-1316"}
   ],
    "urls": [
      {"type": "work", "value": "http://seankelly.biz/"},
      {"type": "home", "value": "http://seankelly.tv/"}
   ]
}
```

Example: The same address book data encoded in JSON

XML vs. JSON (in AJAX Application)

- Most web services provide only XML encoded data.
  - Your server-side script that serves as a proxy to external web services can convert XML-encoded data to JSON format.

- Using `eval()` to parse JSON can be dangerous if the data are coming from an external source.
  - Alternatives – use a JSON parser
    - json.org provides a parser written in JavaScript
    - Some browsers support native JSON parser

Comparison of JSON and XML

- Similarities:
  - Both are human readable
  - Both have very simple syntax
  - Both are hierarchical
  - Both are language independent
- Differences:
  - Syntax is different
  - JSON is less verbose
  - JSON can be parsed by JavaScript's eval method
  - JSON includes arrays
  - Names in JSON must not be JavaScript reserved words
  - XML can be validated

Structural Constraints: Document Type Definitions (DTDs)

The DTD is a grammar defining XML structure

- XML document specifies an associated DTD, plus the root element
- DTD specifies children of the root (and so on)

DTD defines special significance for attributes:

- IDs – special attributes that are analogous to keys for elements
- IDREFs – references to IDs
- IDREFS – represents a list of IDREFs

An Example DTD

Example DTD:

```
<!ELEMENT dblp((mastersthesis | article)*)>
<!ELEMENT mastersthesis(author,title,year,school,committeemember*)>
<!ATTLIST    mastersthesis(mdate          CDATA  #REQUIRED
      key                  ID         #REQUIRED
      advisor              CDATA  #IMPLIED>
<!ELEMENT author(#CDATA)>
…
```

Example use of DTD in XML file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE dblp SYSTEM “my.dtd">
<dblp>…
```

An Example DTD

Occurrence Indicator:

Indicator	**Occurrence**	
(no indicator)	Required	One and only one
?	Optional	None or one
*	Optional, repeatable	None, one, or more
+	Required, repeatable	One or more

An Example DTD

Example DTD:

```
<!ELEMENT dblp((mastersthesis | article)*)>
<!ELEMENT mastersthesis(author,title,year,school,committeemember*)>
<!ATTLIST    mastersthesis(mdate          CDATA  #REQUIRED
      key                  ID         #REQUIRED
      advisor              CDATA  #IMPLIED>
<!ELEMENT author(#CDATA)>
…
```

#REQUIRED: The attribute is required

#IMPLIED: The attribute is not required

#FIXED *some_value*: The attribute is fixed as *some_value*

An Example DTD

Example DTD:

```
<!ELEMENT dblp((mastersthesis | article)*)>
<!ELEMENT mastersthesis(author,title,year,school,committeemember*)>
<!ATTLIST    mastersthesis(mdate          CDATA  #REQUIRED
      key                  ID         #REQUIRED
      advisor              CDATA  #IMPLIED>
<!ELEMENT author(#PCDATA)>
…
```

PCDATA means parsed character data

- The text between the start tag and the end tag WILL be parsed by a parser, and be examined by the parser for entities and markup
- The parser does this because XML elements can contain other elements

CDATA means character data

- This is the text that will NOT be parsed by a parser

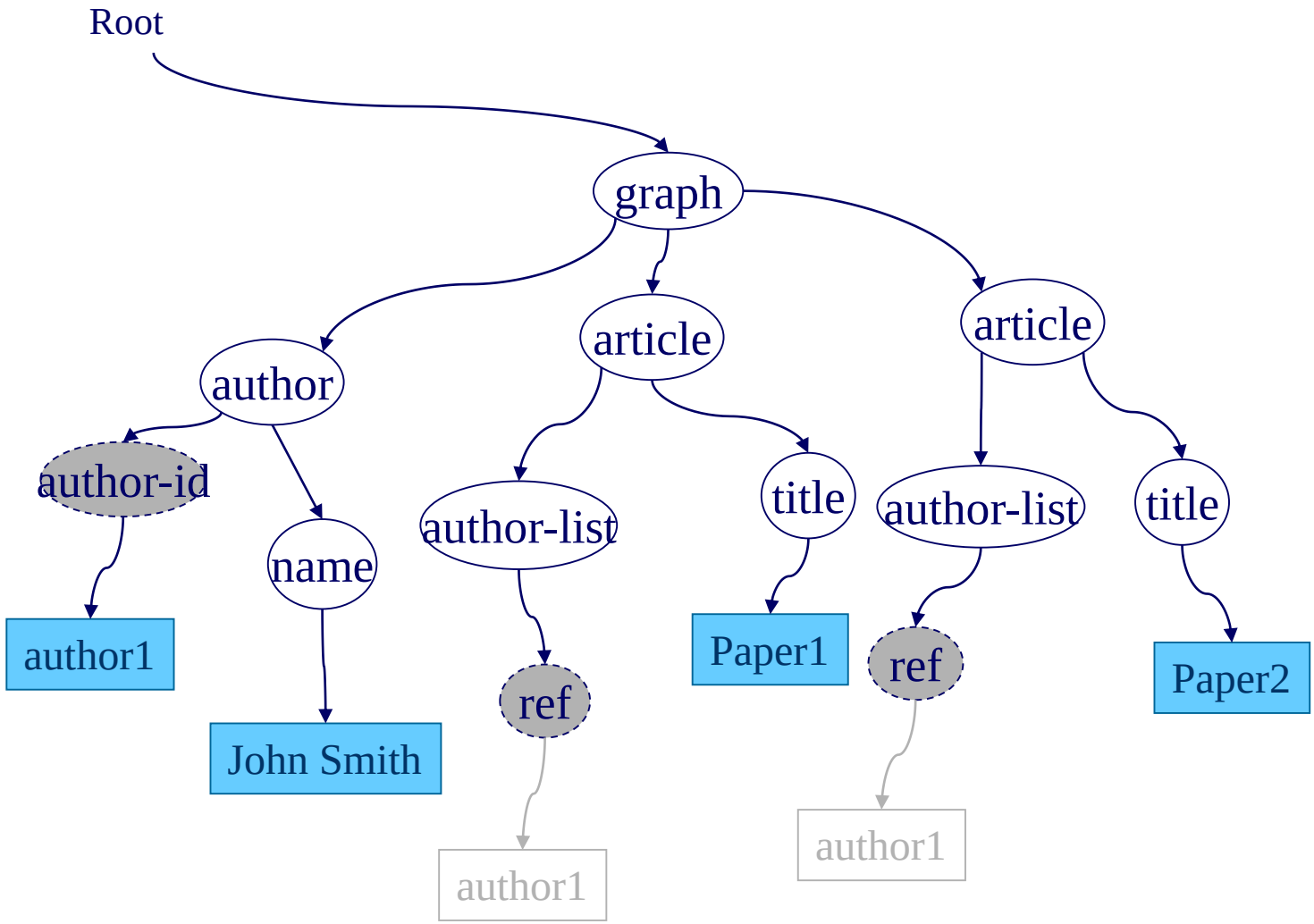
Representing Graphs in XML

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE graph SYSTEM “special.dtd">
<graph>
    <author author-id=“author1”>
        <name>John Smith</name>
    </author>
    <article>
        <author-list ref=“author1” />  <title>Paper1</title>
    </article>
    <article>
        <author-list ref=“author1” />  <title>Paper2</title>
    </article>
…
```

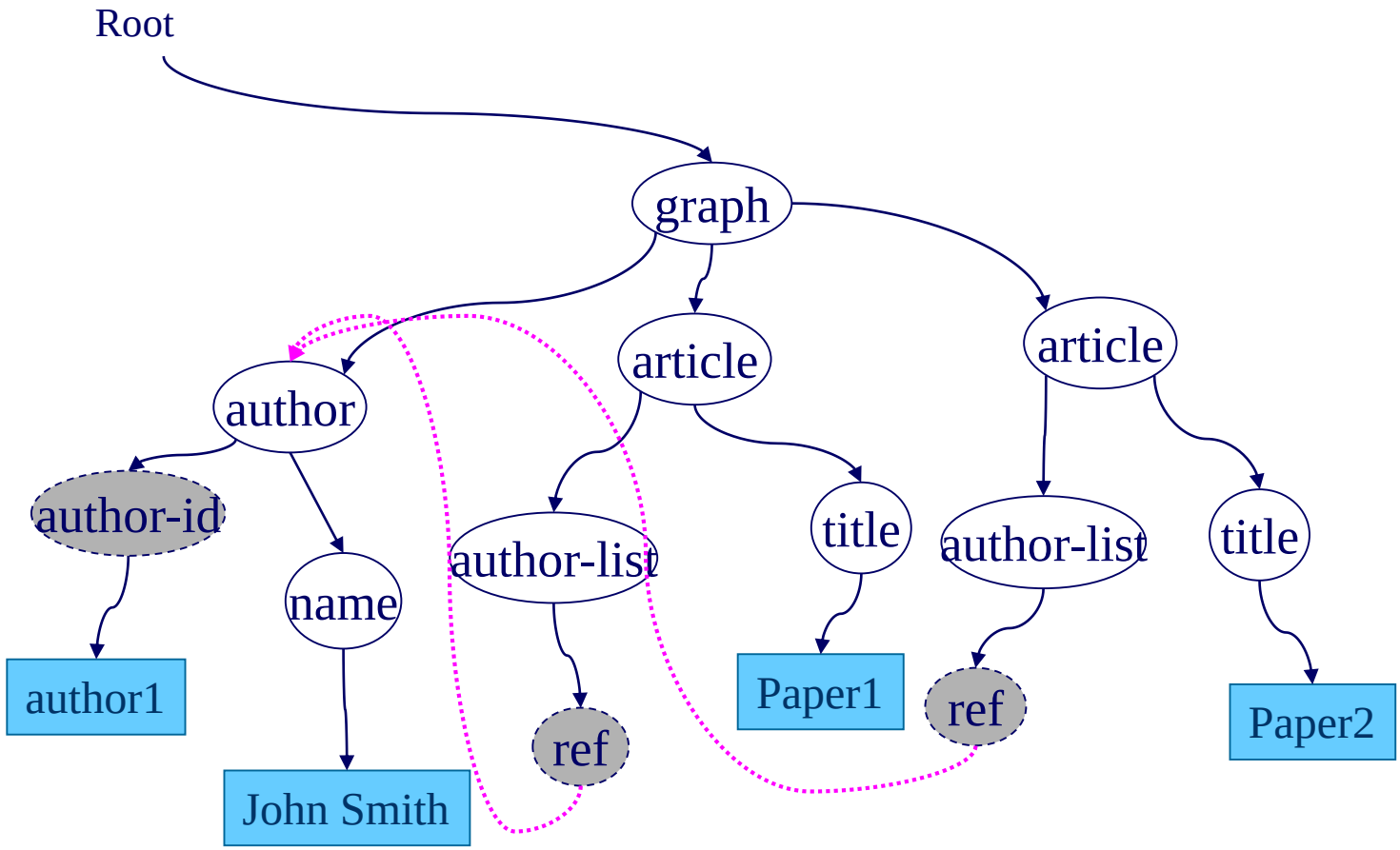
DTD declares ID in attribute

DTD declares IDREFs

Graph Data Model



Graph Data Model



More on attributes

An (opening) tag may contain other (non-ID) attributes, which are typically used to describe the content of an element

```
<entry>
  <word language = "en"> cheese </word>
  <word language = "fr"> fromage </word>
  <meaning> A food made … </meaning>
</entry>
```

Attributes (cont'd)

Another common use for attributes is to express dimension or type

```
<picture>
    <height dim= "cm"> 2400 </height>
    <width dim= "in"> 96 </width>
    <data encoding = "gif" compression = "zip">
        M05-.+C$@02!G96YE<FEC ...
    </data>
</picture>
```

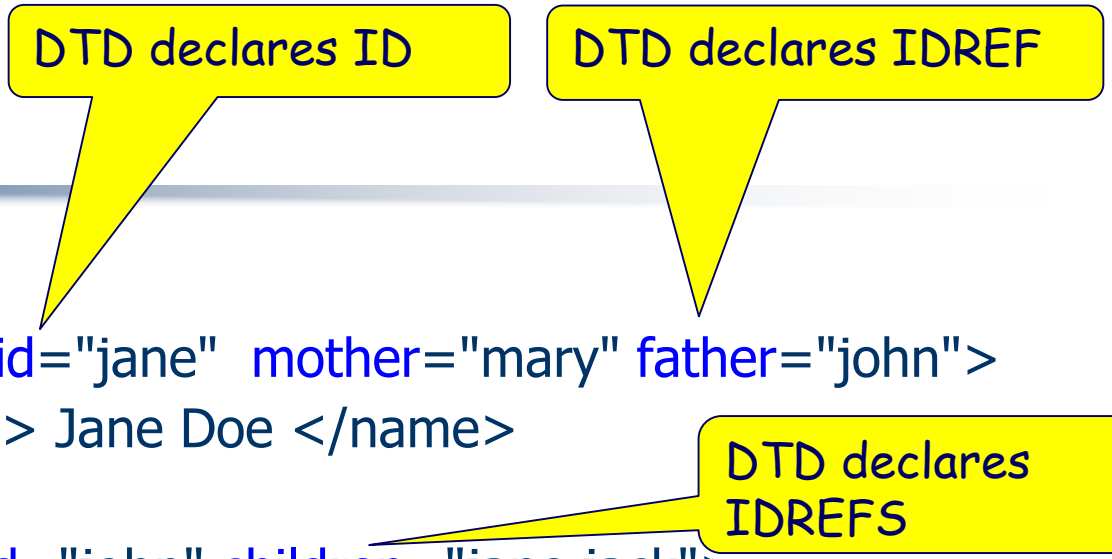
When to use attributes

```
<person ssno= "123 45 6789">
  <name> F. MacNiel </name>
  <email>
     fmacn@dcs.barra.ac.sc
  </email>
  ...
</person>
```

```
<person>
  <ssno> 123 45 6789 </ssno>
  <name> F. MacNiel </name>
  <email>
     fmacn@dcs.barra.ac.sc
  </email>
  ...
</person>
```

The choice between representing data as attributes or as elements is sometimes unclear, taste applies.

Using IDs




```
<family>
    <person  pid="jane"  mother="mary" father="john">
        <name> Jane Doe </name>
    </person>
    <person pid="john" children="jane jack">
        <name> John Doe </name> <mother/>
    </person>
    <person pid="mary" children="jane  jack">
        <name> Mary Doe </name>
    </person>
        <person  pid="jack"  mother=”mary" father="john">
        <name> Jack Doe </name>
    </person>
</family>
```

DTD Example: The Address Book

```
<person>
    <name> MacNiel, John </name>
    <greet> Dr. John MacNiel </greet>
    <addr>1234 Huron Street </addr>
    <addr> Rome, OH 98765 </addr>
    <tel> (321) 786 2543 </tel>
    <fax> (321) 786 2543 </fax>
    <tel> (321) 786 2543 </tel>
    <email> jm@abc.com </email>
</person>
```

Exactly one name

At most one greeting

One or more address lines as needed (in order)

Mixed telephones and faxes

As many as needed

A DTD for the address book

```
<!DOCTYPE addressbook [
  <!ELEMENT addressbook (person*)>
  <!ELEMENT person
      (name, greet?, address+, (fax | tel)*, email*)>
  <!ELEMENT name    (#PCDATA)>
  <!ELEMENT greet   (#PCDATA)>
  <!ELEMENT address (#PCDATA)>
  <!ELEMENT fax     (#PCDATA)>
  <!ELEMENT tel     (#PCDATA)>
  <!ELEMENT email   (#PCDATA)>
]>
```

DTDs Aren't Enough

DTDs capture grammatical structure, but have some drawbacks:

- Not themselves in XML – inconvenient to build tools for them
- Don't capture database datatypes' domains
- IDs aren't a good implementation of keys
  - No element type may have more than one ID attribute specified. The value of an ID attribute must be unique between all values of all ID attributes. Why is this insufficient?
- No way of defining OO-like inheritance

Some things are hard to specify

Each employee element is to contain name, age and ssn elements in some order.

```
<!ELEMENT employee
    ( (name, age, ssn) | (age, ssn, name) |
      (ssn, name, age) | ...
    )>
```

Unfeasible with many fields !

XML Schema

XML Schema

Aims to address the shortcomings of DTDs

Features:

- XML syntax
- Can define keys using XPaths
- Type subclassing that's more complex than in a programming language
  - Programming languages don't consider order of member variables!
  - Subclassing "by extension" and "by restriction"
- … And, of course, domains and built-in datatypes

Why not XML schemas?

- DTDs have been around longer than XSD
  - Therefore they are more widely used
  - Also, more tools support them
- XSD is very verbose, even by XML standards
- More advanced XML Schema instructions can be non-intuitive and confusing

- Nevertheless, XSD is not likely to go away quickly

Referring to a schema

- To refer to a DTD in an XML document, the reference goes *before* the root element:
  - <?xml version="1.0"?>
    <!DOCTYPE rootElement SYSTEM "***url***">
    <rootElement> … </rootElement>
- To refer to an XML Schema in an XML document, the reference goes *in* the root element:
  - <?xml version="1.0"?>
    <rootElement
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    (The XML Schema Instance reference is required)
    xsi:noNamespaceSchemaLocation="***url***.xsd">
    (This is where *your* XML Schema definition can be found)
    …
    </rootElement>

The XSD document

- Since the XSD is written in XML, it can get confusing which we are talking about
- Except for the additions to the root element of our XML data document, the rest of this lecture is about the XSD schema document
- The file extension is .xsd
- The root element is <schema>
- The XSD starts like this:
  - <?xml version="1.0"?>
    <xs:schema xmlns:xs="http://www.w3.rg/2001/XMLSchema">

<schema>

- The <schema> element may have attributes:
  - xmlns:xs="http://www.w3.org/2001/XMLSchema"
    - This is necessary to specify where all our XSD tags are defined
  - elementFormDefault="qualified"
    - This means that all XML elements must be qualified (use a namespace)
    - It is highly desirable to qualify all elements, or problems will arise when another schema is added

“Simple” and “complex” elements

- A “simple” element is one that contains text and nothing else
  - A simple element cannot have attributes
  - A simple element cannot contain other elements
  - A simple element cannot be empty
  - However, the text can be of many different types, and may have various restrictions applied to it
- If an element isn’t simple, it’s “complex”
  - A complex element may have attributes
  - A complex element may be empty, or it may contain text, other elements, or both text and other elements

Defining a simple element

- A simple element is defined as

  <xs:element name="***name***" type="***type***" />

  where:
  - ***name*** is the name of the element
  - the most common values for ***type*** are

    xs:boolean xs:integer
    xs:date xs:string
    xs:decimal xs:time
- Other attributes a simple element may have:
  - default="***default value***" *if no other value is specified*
  - fixed="***value***" *no other value may be specified*

Defining an attribute

- Attributes themselves are always declared as simple types
- An attribute is defined as

  <xs:attribute name="***name***" type="***type***" />

  where:
  - ***name*** and ***type*** are the same as for xs:element
- Other attributes a simple element may have:
  - default="***default value***" *if no other value is specified*
  - fixed="***value***" *no other value may be specified*
  - use="optional" *the attribute is not required (default)*
  - use="required" *the attribute must be present*

Restrictions, or “facets”

- The general form for putting a restriction on a text value is:
  - <xs:element name="*name*"> *(or* xs:attribute*)*

```
<xs:restriction base="type">
    ... the restrictions ...
</xs:restriction>
</xs:element>
```

- For example:
  - <xs:element name="age">

```
<xs:restriction base="xs:integer">
    <xs:minInclusive value="0">
    <xs:maxInclusive value="140">
</xs:restriction>
</xs:element>
```

Restrictions on numbers

- minInclusive -- number must be ≥ the given ***value***
- minExclusive -- number must be > the given ***value***
- maxInclusive -- number must be ≤ the given ***value***
- maxExclusive -- number must be < the given ***value***
- totalDigits -- number must have exactly ***value*** digits
- fractionDigits -- number must have no more than ***value*** digits after the decimal point

Restrictions on strings

- length -- the string must contain exactly ***value*** characters
- minLength -- the string must contain at least ***value*** characters
- maxLength -- the string must contain no more than ***value*** characters
- pattern -- the ***value*** is a regular expression that the string must match
- whiteSpace -- not really a "restriction"--tells what to do with whitespace
  - value="preserve" Keep all whitespace
  - value="replace" Change all whitespace characters to spaces
  - value="collapse" Remove leading and trailing whitespace, and replace all sequences of whitespace with a single space

Enumeration

- An enumeration restricts the value to be one of a fixed set of values
- Example:
  - ```
    <xs:element name="season">
        <xs:simpleType>
            <xs:restriction  base="xs:string">
                <xs:enumeration value="Spring"/>
                <xs:enumeration value="Summer"/>
                <xs:enumeration value="Autumn"/>
                <xs:enumeration value="Fall"/>
                <xs:enumeration value="Winter"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:element>
    ```
    ```

Complex elements

- A complex element is defined as

```
<xs:element   name="name">
   <xs:complexType>
      ... information about the complex type...
   </xs:complexType>
</xs:element>
```

- Example:

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

- <xs:sequence> -- elements must occur in this order
- Remember that attributes are always simple types

Global and local definitions

- Elements declared at the "top level" of a <schema> are available for use throughout the schema
- Elements declared within a xs:complexType are local to that type
- Thus, in

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

  the elements firstName and lastName are only locally declared
- The order of declarations at the "top level" of a <schema> *do not* specify the order in the XML data document

Declaration and use

- So far we've been talking about how to *declare* types, not how to *use* them
- To *use* a type we have declared, use it as the value of type="…"
  - Examples:
    - <xs:element name="student" type="person"/>
    - <xs:element name="professor" type="person"/>
  - Scope is important: you cannot use a type if is local to some other type

xs:sequence

- We've already seen an example of a complex type whose elements must occur in a specific order:
-
```
<xs:element   name="person">
   <xs:complexType>
     <xs:sequence>
       <xs:element  name="firstName"  type="xs:string"
/>
       <xs:element  name="lastName"  type="xs:string"
/>
     </xs:sequence>
   </xs:complexType>
 </xs:element>
```

xs:all

- xs:all allows elements to appear in any order
- ```
  <xs:element   name="person">
     <xs:complexType>
        <xs:all>
           <xs:element  name="firstName"  type="xs:string" />
           <xs:element  name="lastName"  type="xs:string" />
        </xs:all>
     </xs:complexType>
  </xs:element>
  ```
- Despite the name, the members of an xs:all group can occur once or not at all
- You can use minOccurs="0" to specify that an element is optional (default value is 1)
  - In this context, maxOccurs is always 1
  ```

Referencing

- Once you have defined an element or attribute (with name="…"), you can refer to it with ref="…"
- Example:
  - <xs:element name="person">
    <xs:complexType>
    <xs:all>
    <xs:element name="firstName" type="xs:string" />
    <xs:element name="lastName" type="xs:string" />
    </xs:all>
    </xs:complexType>
    </xs:element>
  - <xs:element name="student" ref="person">
  - Or just: <xs:element ref="person">

Text element with attributes

- If a text element has attributes, it is no longer a simple type
  - <xs:element name="population">
    <xs:complexType>
    <xs:simpleContent>
    <xs:extension base="xs:integer">
    <xs:attribute name="year"
    type="xs:integer">
    </xs:extension>
    </xs:simpleContent>
    </xs:complexType>
    </xs:element>

Mixed elements

- Mixed elements may contain both text and elements
- We add mixed="true" to the xs:complexType element
- The text itself is not mentioned in the element, and may go anywhere (it is basically ignored)

-
```
<xs:complexType  name="paragraph"  mixed="true">
    <xs:sequence>
        <xs:element  name="someName”
                     type="xs:anyType"/>
    </xs:sequence>
</xs:complexType>
```

Extensions

- You can base a complex type on another complex type
-
```
<xs:complexType  name="newType">
    <xs:complexContent>
        <xs:extension  base="otherType">
            ...new stuff...
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Predefined string types

- Recall that a simple element is defined as:
  <xs:element name="***name***" type="***type***" />
- Here are a few of the possible string types:
  - xs:string -- a string
  - xs:normalizedString -- a string that doesn't contain tabs, newlines, or carriage returns
  - xs:token -- a string that doesn't contain any whitespace other than single spaces
- Allowable restrictions on strings:
  - enumeration, length, maxLength, minLength, pattern, whiteSpace

Predefined date and time types

- xs:date -- A date in the format ***CCYY-MM-DD***, for example, 2013-11-05
- xs:time -- A date in the format ***hh:mm:ss*** (hours, minutes, seconds)
- xs:dateTime -- Format is ***CCYY-MM-DDThh:mm:ss***
  - The T is part of the syntax
- Allowable restrictions on dates and times:
  - enumeration, minInclusive, minExclusive, maxInclusive, maxExclusive, pattern, whiteSpace

Predefined numeric types

- Here are some of the predefined numeric types:

xs:decimal
xs:byte
xs:short
xs:int
xs:long

xs:positiveInteger
xs:negativeInteger
xs:nonPositiveInteger
xs:nonNegativeInteger

- Allowable restrictions on numeric types:
  - enumeration, minInclusive, minExclusive, maxInclusive, maxExclusive, fractionDigits, totalDigits, pattern, whiteSpace

Schema Example

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="mastersthesis" type="ThesisType"/>
<xsd:complexType name="ThesisType">
    <xsd:attribute name="mdate" type="xsd:date"/>
    <xsd:attribute name="key" type="xsd:string"/>
    <xsd:attribute name="advisor" type="xsd:string"/>
    <xsd:sequence>
        <xsd:element name="author" type="xsd:string"/>
        <xsd:element name="title" type="xsd:string"/>
        <xsd:element name="year" type="xsd:integer"/>
        <xsd:element name="school" type="xsd:string"/>
        <xsd:element name="committeemember" type="CommitteeType"
          minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
```

Designing an XML Schema/DTD

Not as formalized as relational data design

- We can still use ER diagrams to break into entity, relationship sets
- ER diagrams have extensions for "aggregation" – treating smaller diagrams as entities – and for composite attributes
- Note that often we already have our data in relations and need to design the XML schema to export them!

Generally orient the XML tree around the "central" objects

Big decision: element vs. attribute

- Element if it has its own properties, or if you *might* have more than one of them
- Attribute if it is a single property – or perhaps not!

XML Summary

- XML is a semistructured data model that is very flexible, and useful for data exchange.
- A DTD is a "schema language" describing the structure of an XML document
- XPath is the basis for many XML languages, including XML Schema, XQuery, and XSLT
- XQuery is a powerful query language for XML
  - Orthogonal: can always replace a collection with an expression that creates collections
  - Turing Complete due to XQuery functions

References

- JSON
  - http://json.org/
- Speeding Up AJAX with JSON
  - http://www.developer.com/lang/jscript/article.php/3596836