Lecture 3:

Tree Structured Data –
Set Theoretic Query Languages
XML,
Document Type Definitions (DTDs), XML Schema.

2018
Web Data

1. not hard-wired as input data set for a browser/program

2. not a database for a schema as in case of a DBMS

3. Web Data is – (i) Document with objects (ii) Object description tags (iii) Software agent

4. Communication medium $\rightarrow$ Semi-structured data (schema + data) – html, xml, JSON, ....
DOM: Tree-structured Data

- Web: Document (Data) $\rightarrow$ DOM (Tree)
- Database (Relations) $\rightarrow$ Tree
- Rooted, Acyclic with unique path (from root to leaf node) $\rightarrow$ Tree

→ Searching: Breadth first / Depth first

**Example:** [for IIT_DB] Find name at node (Teacher) where sub-node (Course)=“COV885” (Answer – “Bhalla”)
DOM: Tree Structured Data

= Data in Tree form

XML → root (tree)
→ {tree} ; level 1 [ ]; level 2 () ; ....

→ {..[( ) ()] [ ( ) ] ..} Proper Nesting
→ A) rooted Tree
→ B) Unique path from root
→ C) No cycle (tree has no cycle)
Data in XML: Object Model

1. Data (example) \(\rightarrow\) Web Programming

2. xml \(\rightarrow\) Element; Sub-element; Attribute

\[
<\text{name-of-book}> \text{Web Programming} </\text{name-of-book}>
\]

Tag \(\rightarrow\)

\[
<\text{name-of-book} > .. </\text{name-of-book}>
\]

(Opening) Tag can include an attribute \(\rightarrow\)

\[
<\text{name-of-book} \text{ Type=“text-book”} >
\]
Data in XML → meaning of Data

<bookstore>

◆ <book Category = “basic” >
  <title lang = “Italian” > Everyday Italian </title>
  <author> Giada De Laurentiis </author>
  <year> 2005 </year>
  <price> 3000 </price>
◆ </book>

◆ <book> .. .. </book>

◆ .. . . . . .. </bookstore>
Document Object Model (DOM)
http://www.w3schools.com/dom/default.asp

Graph: rooted, unique path from root to leaf, acyclic
Lecture 1

◆ **Documents**: Book Chapter, Report, Book, ..

◆ **Web → Document**

◆ **Documents → have Object Model**

◆ **Example**: Book → **Title, Index, Chapter heading**

◆ **All Books**: DOM (Document Object Model) → Title, index, Chapter 1,2 .., back index.

◆ **Programming**: a) **Client-side** – (DOM Document display)

   HTML, XML, XHTML, CSS, JavaScript, ..

   b) **Server-side**- (Prepare DOM Document)

   PHP, Java/JSP, Javascript,...
Complex Data: Table – Set/bag (represent as Tree)

classification section employee

- c1  s1  e1
- c1  s1  e2
- c1  s2  e3

- <company id="c1">
  - <section id="s1">
    - <employee id="e1"/>
    - <employee id="e2"/>
  - </section>
  - <section id="s2">
    - <employee id="e3"/>
  - </section>
- </company>
Data in Tree (Examples)

<sectionList>
  <section id="s1">
    <company id="c1"/>
    <employee id="e1"/>
    <employee id="e2"/>
  </section>
  <section id="s2">
    <company id="c1"/>
    <employee id="e3"/>
  </section>
</sectionList>

<employeeList>
  <employee id="e1">
    <company id="c1"/>
    <section id="s1"/>
  </employee>
  <employee id="e2">
    <company id="c1"/>
    <section id="s1"/>
  </employee>
  <employee id="e3">
    <company id="c1"/>
    <section id="s2"/>
  </employee>
</employeeList>
Complex Data (case 1) Storing Graph Data $\rightarrow$ set

Attributed Relational Graphs (ARGs)

A
```
A
B
C
D
```

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td>q</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td></td>
<td>s</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td></td>
<td>t</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td></td>
<td>p</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td></td>
<td>r</td>
</tr>
</tbody>
</table>
1. Storing Graph Data in XML → Tree
(rooted tree, acyclic, unique path from root)

- `<node id="A">`
- `<node id="B">
  - `<node id="C">
  - `<node id="D">
  - </node>
  </node>
</node>
Tree Structure Data

- Element node for “C” (Duplicate)
- Rooted tree, acyclic, unique path (root $\rightarrow$ node)
- Subelement nodes $\rightarrow$ Parent-Child

- Tree Structure $\rightarrow$
  A $\rightarrow$ B, C, D
  B $\rightarrow$ C, D
2. Storing Graph Data in XML (Tree) (case 2)

XML with IDs and IDREFS:

```
<node id="A", adj="C D">
  <node id="B">
    <node id="C">
    </node>
    <node id="D">
    </node>
  </node>
</node>
```
Tree Structured Data (case 2)

- Element node for “C” (Not Duplicate)
- Rooted tree, acyclic,
- Unique path (root → node)

- Subelement nodes: Parent → Child
- Node Attributes: Parent → Child
- Tree Structure →

A → B, C, D
B → C, D
2. Storing Graph Data in XML (Tree) (case 2)

XML with IDs and IDREFS:

```
<node id="A">
  <node id="B">
    <node id="C", adj="A">  
    <node id="D", adj="A">  
  </node>
</node>
```

Diagram:

```
A
<br>
B
<br>
C  D
```
1. Storing Graph Data in XML \(\rightarrow\) Graph (case 3) (rooted tree, acyclic, unique path from root)

- `<node id="A">
  - `<node id="B">
    - `<node id="C" adj="A B">
      - `<node id="D" adj="A B">
        - `<node id="C">
          - `<node id="B">
            - `<node id="A">
  - `<node id="D">
  - `<node id="B">
    - `<node id="C">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
  - `<node id="A">
  - `<node id="D">
  - `<node id="B">
2. Storing Graph Data in XML (Graph) (case 2)

XML with IDs and IDREFS:

```
<node id="A", adj="C D">
  <node id="B">
    <node id="C", adj="A">
      </node>
    <node id="D", adj="A">
      </node>
  </node>
</node>
```
Tree structured Data + Graph

- **Element nodes (No Duplicates)**
- **Element → subelement**
  - rooted tree, acyclic,
  - unique path (root → node)
- **Subelement nodes: Parent → Child**
- **Node Attributes: Parent → Child; Child →**
- **Data is Arbitrary Graph Structure Data** →
  - A → B, C, D ; C → A, B ; D → A, B
  - B → C, D
Storing Graph Data as graph

XML (with or without IDREFS)
- Reduces graph database to an XML base
- Use XPath / XQuery engines for attribute queries and structural queries
- Widely supported by a variety of XML parsers

- Costly structure/sub-structure matching
- Needs distinction between-
  (i) IDREF edges (attribute) and
  (ii) hierarchy edges (subelement)
Graph Database Models

◆ “Schema-less” collection of graphs
  • Example: GraphGrep, Daylight ACD, gIndex

◆ Database as a graph
  • Example: SUBDUE

◆ Database with schema and views
  • Example: GRACE
Well-Formed and Valid XML

◆ 1. **Well-Formed XML**
   → has no schema.
   → It allows you to invent your own tags.

◆ 2. **Valid XML**
   → Has a schema
   → DTD (Document Type Definition).

   ![DTD](DTD) → XML file
XML Vocabularies

- Web applications → agree to share common vocabulary (schema), → Banks, Air-line,..
  → to communicate and collaborate
- Many businesses use common (specialized schema)
  - mathematics (MathML)
  - bioinformatics (BSML)
  - human resources (HRML)
  - ...

An XML document consists of

- a prolog
- a number of elements
- an optional epilog (not discussed)
The prolog consists of
- an XML declaration and
- an optional reference to external structuring documents

```xml
<?xml version="1.0" encoding="UTF-16"?>
<!DOCTYPE book SYSTEM "book.dtd">
```
XML Elements

- The “Document” objects (data Content)
  - E.g. **books**, authors, publishers

- An element consists of:
  - an opening tag
  - the content
  - a closing tag  [ May have sub-elements before it ]

```
<lecturer>David Billington</lecturer>
```
XML Elements (2)

- **RULES**
- **Tag names** can be chosen freely.
- The **first character** must be a -
  - letter, an underscore, or a colon
- **No** name may begin with the string "xml" in any combination of cases
  - E.g. "Xml", “xML”
Well-Formed XML

◆ Start the document with a declaration, surrounded by `<?xml ... ?>`.

◆ Normal declaration is:

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

◆ “standalone” = “no DTD provided.”

◆ Balance of document is a root tag surrounding nested tags.
Tags

- Tags are normally matched pairs, as `<FOO>` ... `</FOO>`.
- Unmatched tags also allowed, as `<FOO/>`
- Tags may be nested arbitrarily.
- XML tags are case-sensitive.
Example: Well-Formed XML

```xml
<?xml version = "1.0" standalone = "yes" ?>
<BARS>
  <BAR>
    <NAME>Joe's Bar</NAME>
    <BEER>
      <NAME>Bud</NAME>
      <PRICE>2.50</PRICE>
    </BEER>
    <BEER>
      <NAME>Miller</NAME>
      <PRICE>3.00</PRICE>
    </BEER>
  </BAR>
  ...
</BARS>
```

Root tag

Tags surrounding a BEER element

A NAME subelement

A BEER subelement
Well-Formed XML Documents

- Syntactically correct documents
- Rules:
  - Only one outermost element (called root element)
  - Each element → an opening and closing tag
  - Tags may not overlap
    - `<author><name>Lee Hong</author><name>`
  - Attributes within an element have unique names
  - Element and tag names must be permissible
The Tree Model of XML Documents: An Example

<email>
  <head>
    <from name="Michael Maher"
         address="michaelmaher@cs.gu.edu.au"/>
    <to name="Grigoris Antoniou"
        address="grigoris@cs.unibremen.de"/>
    <subject>Where is your draft?</subject>
  </head>
  <body>
    Grigoris, where is the draft of the paper you promised me last week?
  </body>
</email>
The Tree Model of XML Documents: An Example (2)
The Tree Model of XML Docs

◆ The tree representation of an XML document is an ordered labeled tree:

◆

❖ There is only **one root**
❖ There are **no cycles**
❖ Each **non-root node** has exactly one parent
❖ Each node has a **label**.
❖ The **order** of elements is important
❖ ... but the **order of attributes** is not important
DTD Structure

**DTD contains**  
(a) Tree structure: Element – Sub-element;  
(b) Attributes;  
(c) details of elements,  
(d) occurrences

**Style**

```xml
<!DOCTYPE <root tag> [
  <!ELEMENT <name> (<components>) >
  . . . more elements . . .
]>
```
Purpose of DTD

◆ Define → element and attribute names that may be used

◆ Define → the structure
  ◆ what **values** an attribute may take
  ◆ Tree elements and their sub-elements

◆ With a DTD → the document can be validated
DTD Elements

◆ Description of an element ➔
  Name (tag), and a parenthesized description of any nested tags.
    • Includes order of sub-tags and their multiplicity.

◆ Leaves (text elements) have #PCDATA (Parsed Character DATA) in place of nested tags.
Example: DTD

```xml
<!DOCTYPE BARS [
  <!ELEMENT BARS (BAR*)>
  <!ELEMENT BAR (NAME, BEER+)> 
  <!ELEMENT NAME (#PCDATA)> 
  <!ELEMENT BEER (NAME, PRICE)> 
  <!ELEMENT PRICE (#PCDATA)> ]> 
```

A BARS object has zero or more BAR’s nested within.

A BAR has one NAME and one or more BEER subobjects.

A BEER has a NAME and a PRICE.

NAME and PRICE are text.
Element Descriptions

◆ Subtags must appear in order shown.
◆ A tag may be followed by a symbol to indicate its multiplicity.
  ◆ * = zero or more.
  ◆ + = one or more.
  ◆ ? = zero or one.
◆ Symbol | can connect alternative sequences of tags.
Example: Element Description

A name is an optional title (e.g., “Prof.”), a first name, and a last name, in that order, or it is an IP address:

```xml
<!ELEMENT NAME (TITLE?, FIRST, LAST) | IPADDR>
```
Use of DTD’s

1. Set standalone = “no”.

2. Either:
   a) Include the DTD as a preamble of the XML document, or
   b) Follow DOCTYPE and the <root tag> by SYSTEM and a path to the file (where the DTD can be found → file address or a web url link).
Example: (a)

```xml
<?xml version = "1.0" standalone = "no" ?>
<!DOCTYPE BARS [ 
  <!ELEMENT BARS (BAR*)> 
  <!ELEMENT BAR (NAME, BEER+)> 
  <!ELEMENT NAME (#PCDATA)> 
  <!ELEMENT BEER (NAME, PRICE)> 
  <!ELEMENT PRICE (#PCDATA)> 
]> 

<BARS>
  <BAR><NAME>Joe's Bar</NAME>
    <BEER><NAME>Bud</NAME> <PRICE>2.50</PRICE></BEER>
    <BEER><NAME>Miller</NAME> <PRICE>3.00</PRICE></BEER>
  </BAR>
  ... 
</BARS>
```
Example: (b)

Assume the BARS DTD is in file bar.dtd.

```xml
<?xml version = "1.0" standalone = "no" ?>
<!DOCTYPE BARS SYSTEM "bar.dtd">
<BARS>
  <BAR>
    <NAME>Joe's Bar</NAME>
    <BEER>
      <NAME>Bud</NAME>
      <PRICE>2.50</PRICE>
    </BEER>
    <BEER>
      <NAME>Miller</NAME>
      <PRICE>3.00</PRICE>
    </BEER>
  </BAR>
  <BAR> ...
</BARS>
```
Attributes

◆ Opening tags in XML can have *attributes*.

◆ In a DTD,

```xml
<!ATTLIST E . . . >
```

declares attributes for element $E$, along with its datatype.
Example: Attributes

- Bars can have an attribute `kind`, a character string describing the bar.

```xml
<!ELEMENT BAR (NAME BEER*)>
<!ATTLIST BAR kind CDATA #IMPLIED>
```

Attribute is optional opposite: #REQUIRED

Character string type; no tags
Example: Attribute Use

In a document that allows BAR tags, we might see:

```xml
<BAR kind = "sushi">
  <NAME>Homma’s</NAME>
  <BEER><NAME>Sapporo</NAME>
    <PRICE>5.00</PRICE></BEER>
  ...
</BAR>
```
ID’s and IDREF’s

 Attributes can be pointers from one object to another.
  - Compare to HTML’s NAME = ”foo” and HREF = ”#foo”.

 Allows the structure of an XML document to be a general graph, rather than just a tree.
Creating ID’s

◆ Give an element $E$ an attribute $A$ of type ID.

◆ When using tag $<E>$ in an XML document, give its attribute $A$ a unique value.

◆ Example:

  $<E \quad A = "xyz">$
Creating IDREF’s

To allow elements of type $F$ to refer to another element with an ID attribute, give $F$ an attribute of type IDREF.

Or, let the attribute have type IDREFS, so the $F$-element can refer to any number of other elements.
Example: ID’s and IDREF’s

◆ A new BARS DTD includes both BAR and BEER subelements.
◆ BARS and BEERS have ID attributes name.
◆ BARS have SELLS subelements, consisting of a number (the price of one beer) and an IDREF theBeer leading to that beer.
◆ BEERS have attribute soldBy, which is an IDREFS leading to all the bars that sell it.
Bar elements have name as an ID attribute and have one or more SSELLS subelements.

SELLS elements have a number (the price) and one reference to a beer.

Beer elements have an ID attribute called name, and a soldBy attribute that is a set of Bar names.
Nodes as Semistructured Data

bars.xml

BARS

BAR
name = "JoesBar"

BEER
name = "Bud"
SoldBy = "..."

theBeer = "Bud"

theBeer = "Miller"

PRICE
theBeer = "Bud"

3.00

2.50

Rose = document
Green = element
Gold = attribute
Purple = primitive value
Example: A Document

<BARS>

<BAR name = "JoesBar">
  <SELLS theBeer = "Bud">2.50</SELLS>
  <SELLS theBeer = "Miller">3.00</SELLS>
</BAR> ...

<BEER name = "Bud" soldBy = "JoesBar SuesBar ..." /> ...

</BARS>
Empty Elements

◆ We can do all the work of an element in its attributes.
  ♦ Like BEER in previous example.

◆ Another example: SELLS elements could have attribute \textit{price} rather than a value that is a price.
Example: Empty Element

◆ In the DTD, declare:

```xml
<!ELEMENT SELLS EMPTY>
<!ATTLIST SELLS theBeer IDREF #REQUIRED>
<!ATTLIST SELLS price CDATA #REQUIRED>
```

◆ Example use:

```xml
<SELLS theBeer = "Bud" price = "2.50"/>
```

Note exception to “matching tags” rule
XML Schema

- A more powerful way to describe the structure of XML documents.

- XML-Schema declarations are themselves XML documents.
  - They describe “elements” and the things doing the describing are also “elements.”
Structure of an XML-Schema Document

```xml
<? xml version = ... ?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  ...
</xs:schema>
```

So uses of "xs" within the schema element refer to tags from this namespace.

Defines "xs" to be the namespace described in the URL shown. Any string in place of "xs" is OK.
The `xs:element` Element

◆ Has attributes:

1. **name** = the tag-name of the element being defined.

2. **type** = the type of the element.
   ◆ Could be an XML-Schema type, e.g., `xs:string`.
   ◆ Or the name of a type defined in the document itself.
Example: xs:element

<xs:element name = "NAME"
    type = "xs:string" />

- Describes elements such as
  <NAME>Joe’s Bar</NAME>
Complex Types

To describe elements that consist of subelements, we use `xs:complexType`.
- Attribute `name` gives a name to the type.

Typical subelement of a complex type is `xs:sequence`, which itself has a sequence of `xs:element` subelements.
- Use `minOccurs` and `maxOccurs` attributes to control the number of occurrences of an `xs:element`. 
Example: a Type for Beers

```xml
<xs:complexType name="beerType">
  <xs:sequence>
    <xs:element name="NAME" type="xs:string"
      minOccurs="1" maxOccurs="1" />
    <xs:element name="PRICE" type="xs:float"
      minOccurs="0" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
```

Exactly one occurrence

Like ? in a DTD
An Element of Type beerType

We don’t know the name of the element of this type.
Example: a Type for Bars

```xml
<xs:complexType name="barType">
  <xs:sequence>
    <xs:element name="NAME"
                 type="xs:string"
                 minOccurs="1" maxOccurs="1"/>
    <xs:element name="BEER"
                 type="beerType"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Like * in a DTD
xs:attribute

- xs:attribute elements can be used within a complex type to indicate attributes of elements of that type.

- attributes of xs:attribute:
  - name and type as for xs.element.
  - use = “required” or “optional”.
Example: xs:attribute

```xml
<xs:complexType name = "beerType">
  <xs:attribute name = "name"
    type = "xs:string"
    use = "required" />
  <xs:attribute name = "price"
    type = "xs:float"
    use = "optional" />
</xs:complexType>
```
An Element of This New Type

```
<xxx name = "Bud"
     price = "2.50" />
```

We still don’t know the element name.

The element is empty, since there are no declared subelements.
Restricted Simple Types

- **xs:simpleType** can describe enumerations and range-restricted base types.
- **name** is an attribute
- **xs:restriction** is a subelement.
Restrictions

◆ Attribute **base** gives the simple type to be restricted, e.g., xs:integer.
◆ **xs:{min, max}{Inclusive, Exclusive}** are four attributes that can give a lower or upper bound on a numerical range.
◆ **xs:enumeration** is a subelement with attribute **value** that allows enumerated types.
Example: license Attribute for BAR

```xml
<xs:simpleType name="license">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Full"/>
    <xs:enumeration value="Beer only"/>
    <xs:enumeration value="Sushi"/>
  </xs:restriction>
</xs:simpleType>
```
Example: Prices in Range [1,5)

```xml
<xs:simpleType name = "price">
  <xs:restriction base = "xs:float">
    minInclusive = "1.00"
    maxExclusive = "5.00"
  </xs:restriction>
</xs:simpleType>
```
Keys in XML Schema

◆ An **xs:element** can have an **xs:key** subelement.

◆ **Meaning**: within this element, all subelements reached by a certain **selector** path will have unique values for a certain combination of **fields**.

◆ **Example**: within one BAR element, the **name** attribute of a BEER element is unique.
Example: Key

```xml
<xs:element name = "BAR" ... >
  . . .
  <xs:key name = "barKey">
    <xs:selector xpath = "BEER" />
    <xs:field xpath = "@name" />
  </xs:key>
  . . .
</xs:element>
```

XPath is a query language for XML. All we need to know here is that a path is a sequence of tags separated by `/.`.

And `@` indicates an attribute rather than a tag.
Foreign Keys

An `xs:keyref` subelement within an `xs:element` says that within this element, certain values (defined by selector and field(s), as for keys) must appear as values of a certain key.
Example: Foreign Key

◆ Suppose that we have declared that subelement NAME of BAR is a key for BARS.
  - The name of the key is barKey.

◆ We wish to declare DRINKER elements that have FREQ subelements. An attribute bar of FREQ is a foreign key, referring to the NAME of a BAR.
Example: Foreign Key in XML Schema

<xs:element name = "DRINKERS"

     
<xs:keyref name = "barRef"
      refers = "barKey"
<xs:selector xpath = "DRINKER/FREQ" />
<xs:field xpath = "@bar" />
</xs:keyref>
</xs:element>
Summary

- Use data from Web Documents and Databases

- Complex data $\rightarrow$ Tree or graph

- XML is flexible $\rightarrow$

- Tags are user defined (extensible) Markup for tree and graphs
The XPath/XQuery Data Model

- Corresponding to the fundamental “relation” of the relational model is: sequence of items.
- An item is either:
  1. A primitive value, e.g., integer or string.
  2. A node (defined next).
Principal Kinds of Nodes


2. *Elements* are pieces of a document consisting of some opening tag, its matching closing tag (if any), and everything in between.

3. *Attributes* names that are given values inside opening tags.
Document Nodes

- Formed by `doc(URL)` or `document(URL)`.
- **Example**: `doc(/usr/class/cs145/bars.xml)`
- All XPath (and XQuery) queries refer to a doc node, either explicitly or implicitly.
  - **Example**: key definitions in XML Schema have Xpath expressions that refer to the document described by the schema.
DTD for Running Example

<!DOCTYPE BARS [ 
<!ELEMENT BARS (BAR*, BEER*)> 
<!ELEMENT BAR (PRICE+)>
  <!ATTLIST BAR name ID #REQUIRED>
<!ELEMENT PRICE (#PCDATA)> 
  <!ATTLIST PRICE theBeer IDREF #REQUIRED>
<!ELEMENT BEER EMPTY> 
  <!ATTLIST BEER name ID #REQUIRED>
<!ATTLIST BEER soldBy IDREFS #IMPLIED>
]>
Example Document

<BARS>

<BAR name = "JoesBar">

<PRICE theBeer = "Bud">2.50</PRICE>
<PRICE theBeer = "Miller">3.00</PRICE>

</BAR> ...

</BARS>

<BARS>

<BEER name = "Bud" soldBy = "JoesBar">

SuesBar ...

</BEER>

</BARS>

Document node is all of this, plus the header ( <? xml version... ).
Nodes as Semistructured Data

```
name = "JoesBar"
theBeer = "Miller"
3.00
name = "JoesBar"
theBeer = "Bud"
2.50
name = "Bud"
theBeer = "Bud"

Rose = document
Green = element
Gold = attribute
Purple = primitive
value
```

bars.xml
Paths in XML Documents

- XPath is a language for describing paths in XML documents.
- The result of the described path is a sequence of items.
Path Expressions

◆ Simple path expressions are sequences of slashes (/) and tags, starting with /.
  ◆ Example: /BARS/BAR/PRICE

◆ Construct the result by starting with just the doc node and processing each tag from the left.
Evaluating a Path Expression

◆ Assume the first tag is the root.
  ◆ Processing the doc node by this tag results in a sequence consisting of only the root element.

◆ Suppose we have a sequence of items, and the next tag is $\chi$.
  ◆ For each item that is an element node, replace the element by the subelements with tag $\chi$. 
Example: /BARS

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer = "Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> ...
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ... "/> ...
</BARS>

One item, the BARS element
Example: /BARS/BAR

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer ="Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> ...
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ...">
  
  </BAR> ...
  </BARS>

This BAR element followed by all the other BAR elements
Example: /BARS/BAR/PRICE

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer ="Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> ...
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ..."/>
  ...
</BARS>

These PRICE elements followed by the PRICE elements of all the other bars.
Attributes in Paths

◆ Instead of going to subelements with a given tag, you can go to an attribute of the elements you already have.

◆ An attribute is indicated by putting @ in front of its name.
Example:
/BARS/BAR/PRICE/@theBeer

<BARS>
  <BAR name = "JoesBar">  
    <PRICE theBeer = "Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> ...
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ..."/> ...
</BARS>

These attributes contribute "Bud" "Miller" to the result, followed by other theBeer values.
**Remember:** Item Sequences

- Until now, all item sequences have been sequences of elements.
- When a path expression ends in an attribute, the result is typically a sequence of values of primitive type, such as strings in the previous example.
Paths that Begin Anywhere

If the path starts from the document node and begins with //X, then the first step can begin at the root or any subelement of the root, as long as the tag is X.
Example: //PRICE

<BARS>
  <BAR name = "JoesBar">
    <PRICE theBeer ="Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  </BAR> ...
  <BEER name = "Bud" soldBy = "JoesBar SuesBar ..."/>
</BARS>

These PRICE elements and any other PRICE elements in the entire document
Wild-Card *

◆ A star (*) in place of a tag represents any one tag.

◆ Example: /*/*/*/PRICE represents all price objects at the third level of nesting.
Example: /BARS/*

This BAR element, all other BAR elements, the BEER element, all other BEER elements

<BARS>

<BAR name = "JoesBar">
  <PRICE theBeer = "Bud">2.50</PRICE>
  <PRICE theBeer = "Miller">3.00</PRICE>
</BAR> ...

<BEER name = "Bud" soldBy = "JoesBar SuesBar ... ">
  ...
</BARS>
Selection Conditions

- A condition inside [...] may follow a tag.
- If so, then only paths that have that tag and also satisfy the condition are included in the result of a path expression.
Example: Selection Condition

◆ /BARS/BAR/PRICE[$ < 2.75]

<BARS>

<BAR name = "JoesBar">

<PRICE theBeer = "Bud">2.50</PRICE>

<PRICE theBeer = "Miller">3.00</PRICE>

</BAR> ...

The condition that the PRICE be < $2.75 makes this price but not the Miller price part of the result.

The current element.
Example: Attribute in Selection

◆ /BARS/BAR/PRICE[@theBeer = "Miller"]

<BARS>

  <BAR name = "JoesBar">

    <PRICE theBeer = "Bud">2.50</PRICE>
    <PRICE theBeer = "Miller">3.00</PRICE>
  
  </BAR>

</BARS> ...

Now, this PRICE element is selected, along with any other prices for Miller.
Axes

◆ In general, path expressions allow us to start at the root and execute steps to find a sequence of nodes at each step.
◆ At each step, we may follow any one of several axes.
◆ The default axis is child:: --- go to all the children of the current set of nodes.
Example: Axes

◗ /BARS/BEER is really shorthand for /BARS/child::BEER.

◗ @ is really shorthand for the attribute:: axis.

❖ Thus, /BARS/BEER[@name = "Bud"] is shorthand for /BARS/BEER[attribute::name = "Bud"]
More Axes

Some other useful axes are:

1. **parent::** = parent(s) of the current node(s).

2. **descendant-or-self::** = the current node(s) and all descendants.
   - Note: // is really shorthand for this axis.

3. **ancestor::, ancestor-or-self,** etc.

4. **self** (the dot).
Summary

- Data on the web – shared by Applications and browsers
- Semi-structured → DTD + tagged contents
- Volume → Documents and Databases
- Storage and Query Systems