XML

Structured, Semistructured, and Unstructured Data

- **Structured data**
  - Represented in a strict format
  - Example: information stored in databases
- **Semistructured data**
  - Has a certain structure
  - Not all information collected will have identical structure

Structured, Semistructured, and Unstructured Data (cont'd.)

- Schema information mixed in with data values
- **Self-describing data**
- May be displayed as a directed graph
  - **Labels** or **tags** on directed edges represent:
    - Schema names
    - Names of attributes
    - Object types (or entity types or classes)
    - Relationships

Structured, Semistructured, and Unstructured Data (cont'd.)

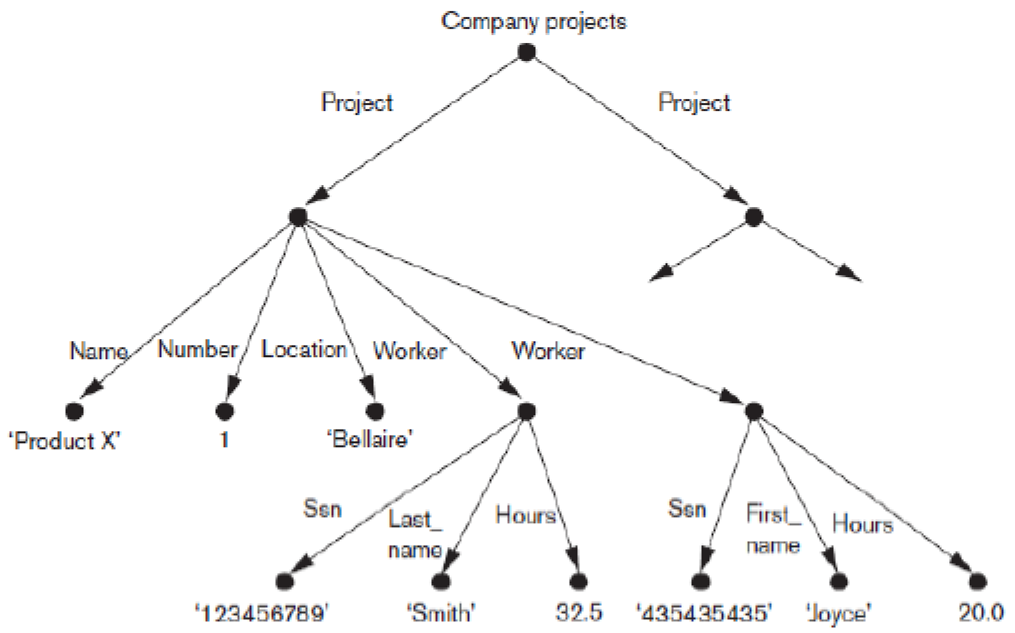

Figure 12.1
Representing semistructured data as a graph.

Structured, Semistructured, and Unstructured Data (cont'd.)

- **Unstructured data**
  - Limited indication of the of data document that contains information embedded within it
- **HTML tag**
  - Text that appears between angled brackets: `<...>`
- **End tag**
  - Tag with a slash: `</...>`

Introduction

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

XML Introduction (Cont.)

- 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>
```

XML: Motivation

- Data interchange is critical in today's networked world
  - Examples:
    - Banking: funds transfer
    - Order processing (especially inter-company orders)
    - Scientific data
      - Chemistry: ChemML, …
      - Genetics: BSML (Bio-Sequence Markup Language), …
  - Paper flow of information between organizations is being replaced by electronic flow of information
- Each application area has its own set of standards for representing information
- XML has become the basis for all new generation data interchange formats

XML Motivation (Cont.)

- 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)
- 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
  - Allows nested structures
  - Wide acceptance, not only in database systems, but also in browsers, tools, and applications

XML Documents, DTD, and XML Schema

- **Well formed**
  - Has **XML declaration**
    - Indicates version of XML being used as well as any other relevant attributes
  - Every element must matching pair of start and end tags
    - Within start and end tags of parent element
- **DOM** (Document Object Model)
  - Manipulate resulting tree representation corresponding to a well-formed XML document

XML Documents, DTD, and XML Schema (cont'd.)

- **SAX** (Simple API for XML)
  - Processing of XML documents on the fly
    - Notifies processing program through callbacks whenever a start or end tag is encountered
  - Makes it easier to process large documents
  - Allows for **streaming**

XML Documents, DTD, and XML Schema (cont'd.)

- **Valid**
  - Document must be well formed
  - Document must follow a particular schema
  - Start and end tag pairs must follow structure specified in separate XML **DTD (Document Type Definition)** file or XML schema file

Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with <*tagname*> and ending with matching </*tagname*>
- Elements must be properly nested
  - Proper nesting
    - <course> … <title> …. </title> </course>
  - Improper nesting
    - <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.
- Every document must have a single top-level element

Example of Nested Elements

```
<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>
```

Motivation for Nesting

- Nesting of data is useful in data transfer
  - Example: elements representing *item* nested within an *itemlist* element
- Nesting is not supported, or discouraged, in relational databases
  - With multiple orders, customer name and address are stored redundantly
  - normalization replaces nested structures in each order by foreign key into table storing customer name and address information
  - Nesting is supported in object-relational databases
- But nesting is appropriate when transferring data
  - External application does not have direct access to data referenced by a foreign key

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">
```

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>
```

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)**
    - Widely used
  - **XML Schema**
    - Newer, increasing use

XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - Typing of values
    - E.g. integer, string, etc
    - Also, constraints on min/max values
  - User-defined, complex types
  - Many more features, including
    - uniqueness and foreign key constraints, inheritance
- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
- XML Scheme is integrated with namespaces
- BUT: XML Schema is significantly more complicated than DTDs.

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
- 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>
```

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

```
<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

Example: a Type for Bars

```
<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>
```

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

```
<xs:complexType name = ”beerType”>
  <xs:attribute name = ”name”
     type = ”xs:string”
     use = ”required” />
  <xs:attribute name = ”price”
     type = ”xs:float”
     use = ”optional” />
</xs:complexType>
```

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

```
<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)

```
<xs:simpleType name = "price">
  <xs:restriction
     base = "xs:float"
     minInclusive = "1.00"
     maxExclusive = "5.00" />
</xs:simpleType>
```

Availables Constraints

Constraint	Description
enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero
length	Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
maxExclusive	Specifies the upper bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
maxLength	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
minLength	Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
pattern	Defines the exact sequence of characters that are acceptable
totalDigits	Specifies the exact number of digits allowed. Must be greater than zero
whiteSpace	Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

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

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

And @ indicates an attribute rather than a tag.

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

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>
```

Querying and Transforming XML Data

- Translation of information from one XML schema to another
- Querying on XML data
- Above two are closely related, and handled by the same tools
- Standard XML querying/translation languages
  - XPath
    - Simple language consisting of path expressions
  - XSLT
    - Simple language designed for translation from XML to XML and XML to HTML
  - XQuery
    - An XML query language with a rich set of features

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

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.

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 evaluated on the university-3 data we saw earlier returns

  ```
  <name>Srinivasan</name>
  <name>Brandt</name>
  ```
- E.g. /university-3/instructor/name/text( )
  returns the same names, but without the enclosing tags

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 *X*.
  - For each item that is an element node, replace the element by the subelements with tag *X*.

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 …"/> …
</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 followe
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 BA
elements, the BEER element, a
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 current element.

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

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> …
```

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).

XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
- XQuery is derived from the Quilt query language, which itself borrows from SQL, XQL and XML-QL
- XQuery uses a

  for … let … where … order by …result …

  syntax

  for ⇔ SQL **from**

  where ⇔ SQL **where**

  order by ⇔ SQL **order by**

  result ⇔ SQL **select**

  let allows temporary variables, and has no equivalent in SQL

FLWR Expressions

1. One or more for and/or let clauses.
2. Then an optional where clause.
3. A return clause.

Semantics of FLWR Expressions

- Each for creates a loop.
  - let produces only a local definition.
- At each iteration of the nested loops, if any, evaluate the where clause.
- If the where clause returns TRUE, invoke the return clause, and append its value to the output.

FOR Clauses

for <variable> in <expression>, . . .

- Variables begin with $.
- A for-variable takes on each item in the sequence denoted by the expression, in turn.
- Whatever follows this for is executed once for each value of the variable.

Example: FOR

"Expand the enclosed string by replacing variables and path exps. by their values."

```
for $beer in
   document("bars.xml")/BARS/BEER/@name
return
   <BEERNAME> {$beer} </BEERNAME>
```

- $beer ranges over the name attributes of all beers in our example document.
- Result is a sequence of BEERNAME elements:
  <BEERNAME>Bud</BEERNAME>
  <BEERNAME>Miller</BEERNAME> . . .

Use of Braces

- When a variable name like $x, or an expression, could be text, we need to surround it by braces to avoid having it interpreted literally.
  - Example: <A>$x</A> is an A-element with value "$x", just like <A>foo</A> is an A-element with "foo" as value.

Use of Braces --- (2)

- But `return $x` is unambiguous.
- You cannot return an untagged string without quoting it, as `return "$x"`.

LET Clauses

let <variable> := <expression>, . . .

- Value of the variable becomes the *sequence* of items defined by the expression.
- Note let does not cause iteration; for does.

Example: LET

```
let $d := document("bars.xml")
let $beers := $d/BARS/BEER/@name
return
   <BEERNAMES> {$beers} </BEERNAMES>
```

- Returns one element with all the names of the beers, like:

```
<BEERNAMES>Bud Miller …</BEERNAMES>
```

Order-By Clauses

- FLWR is really FLWOR: an order-by clause can precede the return.
- Form: order by <expression>
  - With optional ascending or descending.
- The expression is evaluated for each assignment to variables.
- Determines placement in output sequence.

Example: Order-By

- List all prices for Bud, lowest first.

```
let $d := document("bars.xml")
for $p in $d/BARS/BAR/PRICE[@theBeer="Bud"]
order by $p
return $p
```

Generates bindings for $p to PRICE elements.

Order those bindings by the values inside the elements (automatic coersion).

Each binding is evaluated for the output. The result is a sequence of PRICE elements.

Predicates

- Normally, conditions imply existential quantification.
- Example: /BARS/BAR[@name] means "all the bars that have a name."
- Example: /BARS/BEER[@soldAt = "JoesBar"] gives the set of beers that are sold at Joe's Bar.

Example: Comparisons

- Let us produce the PRICE elements (from all bars) for all the beers that are sold by Joe's Bar.
- The output will be BBP elements with the names of the bar and beer as attributes and the price element as a subelement.

Strategy

1. Create a triple for-loop, with variables ranging over all BEER elements, all BAR elements, and all PRICE elements within those BAR elements.
2. Check that the beer is sold at Joe's Bar and that the name of the beer and theBeer in the PRICE element match.
3. Construct the output element.

The Query

```
let $bars = doc("bars.xml")/BARS
for $beer in $bars/BEER
for $bar in $bars/BAR
for $price in $bar/PRICE
where $beer/@soldAt = "JoesBar" and
  $price/@theBeer = $beer/@name
return <BBP bar = {$bar/@name} beer
  = {$beer/@name}>{$price}</BBP>
```

True if "JoesBar" appears anywhere in the sequence

Strict Comparisons

- To require that the things being compared are sequences of only one element, use the Fortran comparison operators:
  - eq, ne, lt, le, gt, ge.
- Example: $beer/@soldAt eq "JoesBar" is true only if Joe's is the only bar selling the beer.

Comparison of Elements and Values

- When an element is compared to a primitive value, the element is treated as its value, if that value is atomic.
- Example: `/BARS/BAR[@name=”JoesBar”]/PRICE[@theBeer=”Bud”] eq ”2.50”`

  is true if Joe charges $2.50 for Bud.

Comparison of Two Elements

- It is insufficient that two elements look alike.
- Example:

```
/BARS/BAR[@name=”JoesBar”]/
PRICE[@theBeer=”Bud”] eq
/BARS/BAR[@name=”SuesBar”]/
PRICE[@theBeer=”Bud”]
```

is false, even if Joe and Sue charge the same for Bud.

Comparison of Elements – (2)

- For elements to be equal, they must be the same, physically, in the implied document.
- Subtlety: elements are really pointers to sections of particular documents, not the text strings appearing in the section.

Getting Data From Elements

- Suppose we want to compare the values of elements, rather than their location in documents.
- To extract just the value (e.g., the price itself) from an element *E*, use data(*E*).

Example: data()

- Suppose we want to modify the return for "find the prices of beers at bars that sell a beer Joe sells" to produce an empty BBP element with price as one of its attributes.

```
return <BBP bar = {$bar/@name} beer
= {$beer/@name} price =
{data($price)} />
```

Eliminating Duplicates

- Use function `distinct-values` applied to a sequence.
- Subtlety: this function strips tags away from elements and compares the string values.
  - But it doesn't restore the tags in the result.

Example: All the Distinct Prices

```
return distinct-values(
    let $bars = doc("bars.xml")
    return $bars/BARS/BAR/PRICE
)
```

Remember: XQuery is
an expression language.
A query can appear any
place a value can.

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

Application Program Interface

- There are two standard application program interfaces to XML data:
  - **SAX** (Simple API for XML)
    - Based on parser model, user provides event handlers for parsing events
      - E.g. start of element, end of element
  - **DOM** (Document Object Model)
    - **XML** data is parsed into a tree representation
    - Variety of functions provided for traversing the DOM tree
    - E.g.: Java DOM API provides Node class with methods
      getParentNode( ), getFirstChild( ), getNextSibling( )
      getAttribute( ), getData( ) (for text node)
      getElementsByTagName( ), …
    - Also provides functions for updating DOM tree

SQL/XML

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

```
<university>
  <department>
    <row>
      <dept name> Comp. Sci. </dept name>
      <building> Taylor </building>
      <budget> 100000 </budget>
    </row>
    .... more rows if there are more output tuples ...
  </department>
  ... other relations ..
</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
```

- Xmlagg creates a forest of XML elements

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

XML Applications

- Storing and exchanging data with complex structures
  - E.g. Open Document Format (ODF) format standard for storing Open Office and Office Open XML (OOXML) format standard for storing Microsoft Office documents
  - Numerous other standards for a variety of applications
    - ChemML, MathML
- Standard for data exchange for Web services
  - remote method invocation over HTTP protocol
  - More in next slide
- Data mediation
  - Common data representation format to bridge different systems

Web Services

- The Simple Object Access Protocol (SOAP) standard:
  - Invocation of procedures across applications with distinct databases
  - XML used to represent procedure input and output
- A *Web service* is a site providing a collection of SOAP procedures
  - Described using the Web Services Description Language (WSDL)
  - Directories of Web services are described using the Universal Description, Discovery, and Integration (UDDI) standard

Implementación de XML en Oracle 1

- **XMLType**
- XMLType is a native server datatype that allows the database to understand that a column or table contains XML. XMLType also provides methods that allow common operations such as XML schema validation and XSL transformations on XML content.You can use the XMLType data-type like any other datatype. For example, you can use XMLType when:
- Creating a column in a relational table
- Declaring PL/SQL variables
- Defining and calling PL/SQL procedures and functions
- Since XMLType is an object type, you can also create a *table* of XMLType. By default, an XMLType table or column can contain any well-formed XML document.

Implementación de XML en Oracle 2

```
SELECT extractValue(object_value,'/PurchaseOrder/Reference')
    "REFERENCE"
FROM PURCHASEORDER
WHERE
    existsNode(object_value,'/PurchaseOrder[SpecialInstructions="
    Expidite"]') = 1;
```

Implementación de XML en Oracle 3

```
SELECT
    p.object_value.extract('/PurchaseOrder/Requestor/text()').getStrin
    gVal() NAME, count(*)
FROM PURCHASEORDER p
WHERE p.object_value.existsNode (
    '/PurchaseOrder/ShippingInstructions[ora:contains(address/text(),"
    Shores")>0]', 'xmlns:ora="http://xmlns.oracle.com/xdb' ) = 1 AND
    p.object_value.extract('/PurchaseOrder/Requestor/text()').getStrin
    gVal() like '%ll%'
GROUP BY
    p.object_value.extract('/PurchaseOrder/Requestor/text()').getStrin
    gVal();
```

Implementación de XML en Oracle 4

```
UPDATE PURCHASEORDER
SET object_value =
   updateXML(object_value,'/PurchaseOrder/Actio
   ns/Action[1]/User/text()','SKING')
WHERE
   existsNode(object_value,'/PurchaseOrder[Refer
   ence="SBELL-2002100912333601PDT"]') = 1
```

Presentación

- Esta presentación fue armada utilizando, además de material propio, material contenido en los manuales de Oracle y material provisto por los siguientes autores
- Siblberschat, Korth, Sudarshan - Database Systems Concepts, 6th Ed., Mc Graw Hill, 2010
- García Molina/Ullman/Widom - Database Systems: The Complete Book, 2nd Ed.,Prentice Hall, 2009
- Elmasri/Navathe - Fundamentals of Database Systems, 6th Ed., Addison Wesley, 2011