VirginiaTech

CS 4604: Introduction to Database Management Systems

Prakash, Spring 2014

Substitute Lecture by

Prof. Eli Tilevich

Computer Science Department

Lecture #14: Semi-Structured Data and XML

Framework

1. *Information Integration* : Making databases from various places work as one.
2. *Semistructured Data* : A (not really) new data model designed to cope with problems of information integration.
3. *XML* : A standard language for describing semistructured data schemas and representing data.

The Information-Integration Problem

- Related data exists in many places and could, in principle, work together.
- But different databases differ in:

1. Model (relational, object-oriented?).
2. Schema (normalized/unnormalized?).
3. Terminology: are consultants employees? Retirees? Subcontractors?
4. Conventions (meters versus feet?).

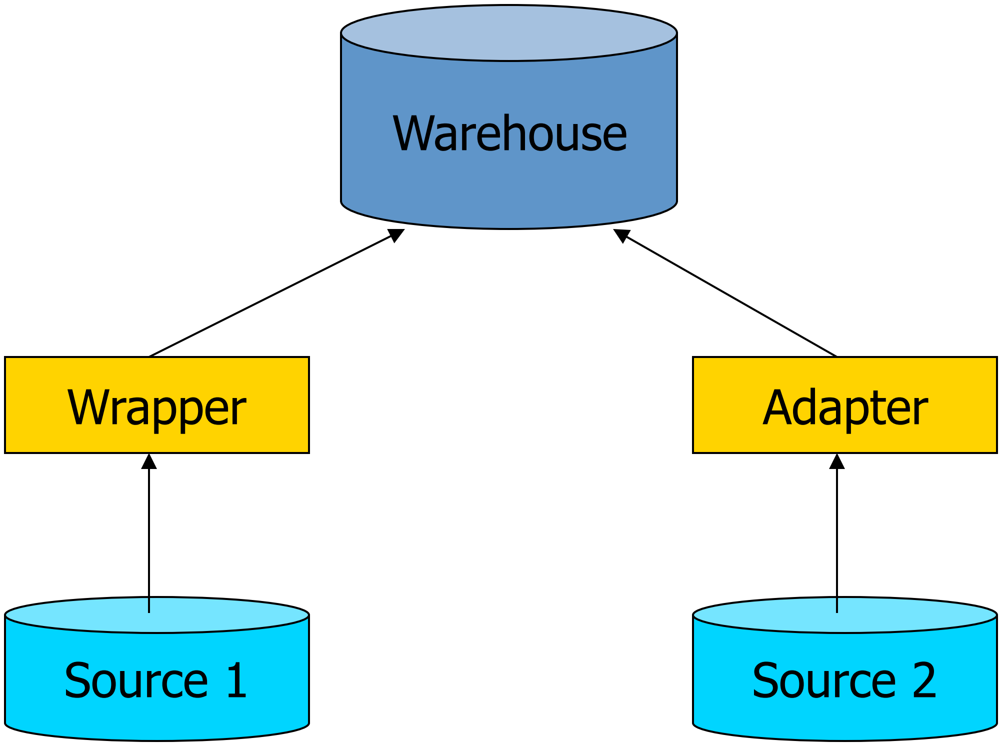
Example

- Every bar in Bburg has a database.
  - One may use a relational DBMS; another keeps the menu in an MS-Word document.
  - One stores the phones of distributors, another does not.
  - One distinguishes ales from other beers, another doesn't.
  - One counts beer inventory by bottles, another by cases.

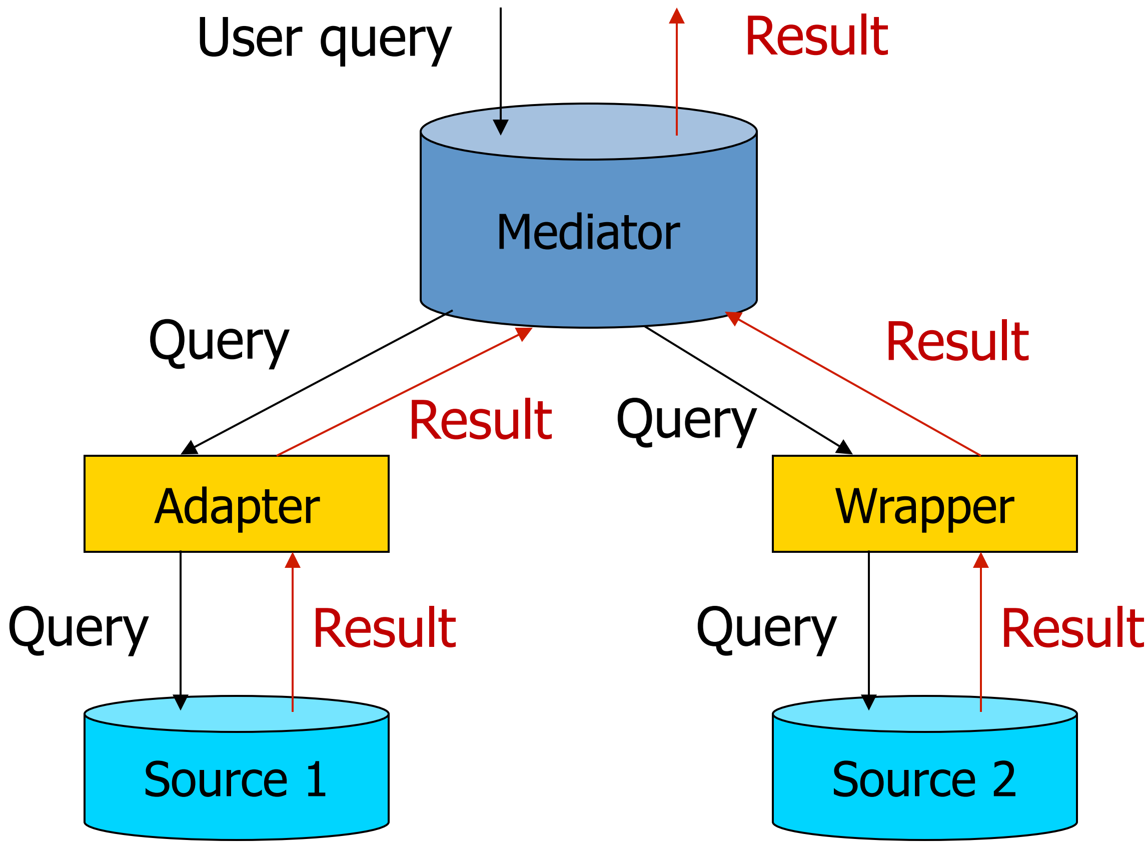
Two Approaches to Integration

1. *Warehousing* : Make copies of the data sources at a central site and transform it to a common schema.
   – Reconstruct data daily/weekly, but do not try to keep it more up-to-date than that.
2. *Mediation* : Create a view of all sources, as if they were integrated.
   – Answer a view query by translating it to terminology of the sources and querying them.

Warehouse Diagram



A Mediator



User query
Result
Mediator
Query
Result
Query
Result
Adapter
Wrapper
Query
Result
Query
Result
Source 1
Source 2

Semistructured Data

- Purpose: represent data from independent sources more flexibly than either relational or object-oriented models.

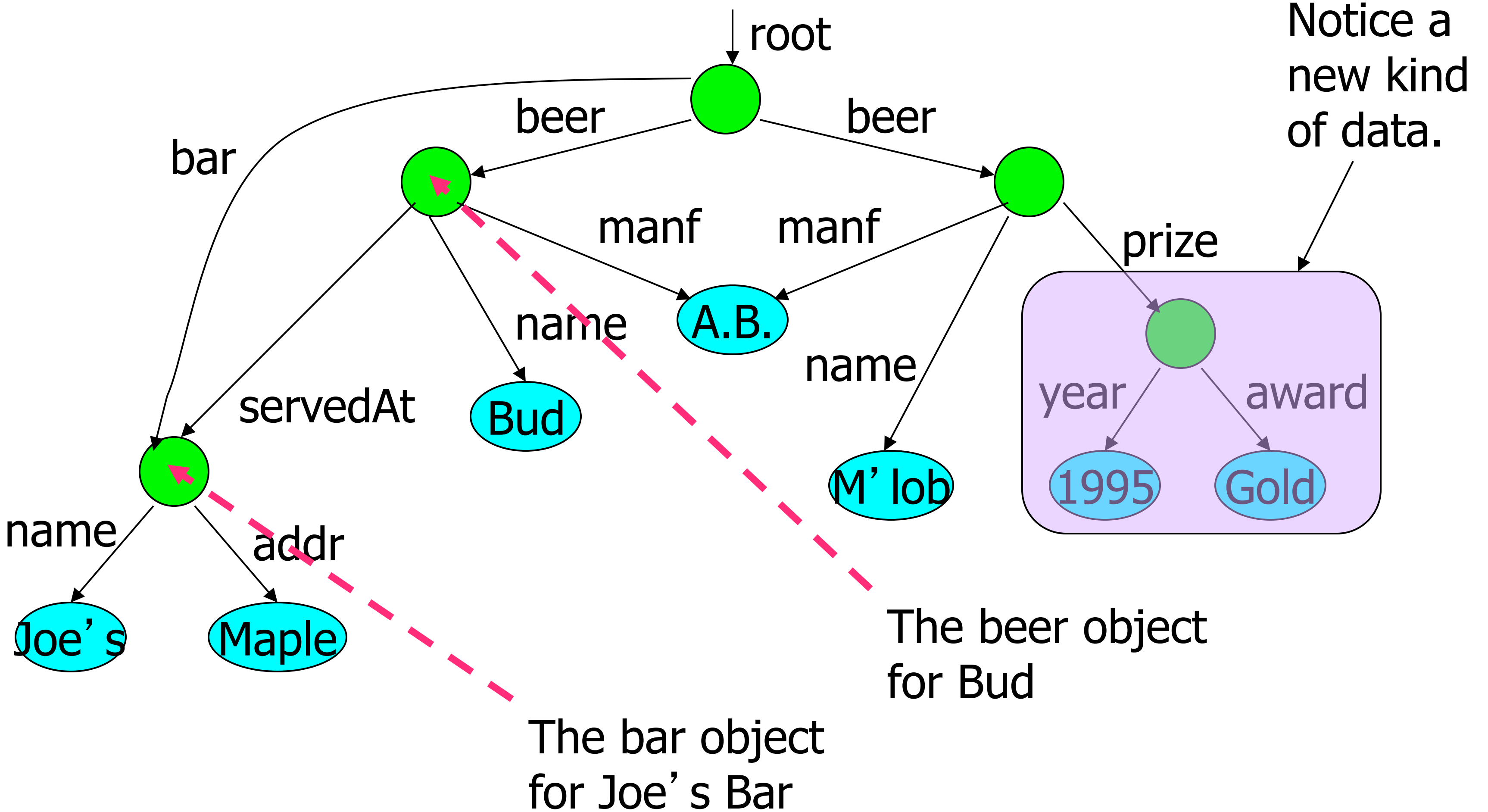
- Think of objects, but with the type of each object its own business, not that of its "class."

- Labels to indicate meaning of substructures.

Graphs of Semistructured Data

- Nodes = objects.
- Labels on arcs (attributes, relationships).
- Atomic values at leaf nodes (nodes with no arcs out).
- Flexibility: no restriction on:
  – Labels out of a node.
  – Number of successors with a given label.

Example: Data Graph



XML

- XML = EXtensible Markup Language.
- While HTML uses tags for formatting (e.g., "italic"), XML uses tags for semantics (e.g., "this is an address").
- Key idea: create tag sets for a domain (e.g., genomics), and translate all data into properly tagged XML documents.

Well-Formed and Valid XML

- *Well-Formed XML* allows you to invent your own tags.
  - Similar to labels in semistructured data.
- *Valid XML* involves a DTD (Document Type Definition), which limits the labels and gives a grammar for their use.

Is a Well-formed Document Valid?

- An XML document is said to be *well-formed* if it follows all of the "rules" of XML, such as proper nesting and attribute use, so by definition all XML documents are well-formed.

- A *valid* document, on the other hand, is one that is not only well-formed, but also follows the restrictions set out in a specific grammar, typically specified in a Document Type Definition (DTD) or some form of XML Schema.

Is a Wellformed Document Valid?

- An example of a document that is well-formed but not valid based upon the XHTML grammar.

```
<body>
  <p>Example of Well-formed HTML</p>
  <head>
    <title>Example</title>
  </head>
  <zorko>What is this?</zorko>
</body>
```

Why?

HTML vs. XML

- In the case of HTML, browsers have been taught how to ignore invalid HTML such as the <zorko> element and generally do their best when dealing with badly placed HTML elements.

- The XML processor, on the other hand, can not tell us which elements and attributes are valid. As a result we need to define the XML markup we are using. To do this, we need to define the markup language's grammar.

Well-Formed XML

- Start the document with a *declaration*, surrounded by <? … ?> .
- Normal declaration is:

```
<? XML VERSION = "1.0" STANDALONE = "yes" ?>
```

  - "Standalone" = "no DTD provided."
- Balance of document is a *root tag* surrounding nested tags.

Tags

- Tags, as in HTML, are normally matched pairs, as <FOO> … </FOO> .
- Tags may be nested arbitrarily.
- Tags requiring no matching ender, like <P> in HTML, are also permitted.

Example: Well-Formed 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>
   <BAR> …
</BARS>
```

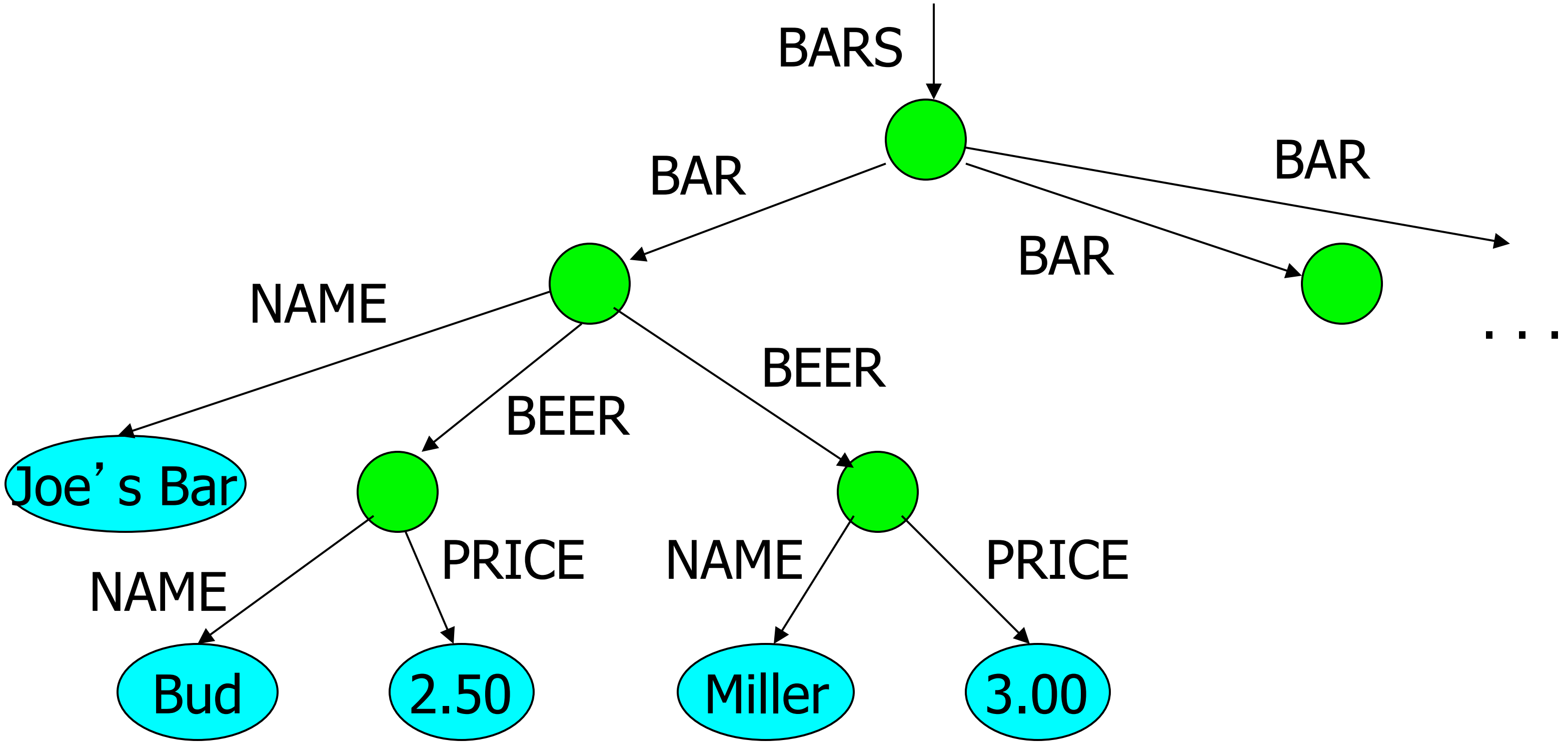
XML and Semistructured Data

- Well-Formed XML with nested tags is exactly the same idea as trees of semistructured data.

- We shall see that XML also enables nontree structures, as does the semistructured data model.

Example

- The <BARS> XML document is:



Document Type Definitions

- Essentially a context-free grammar for describing XML tags and their nesting.
- Each domain of interest (e.g., electronic components, bars-beers-drinkers) creates one DTD that describes all the documents this group will share.

DTD Structure

```
<!DOCTYPE <root tag> [
  <!ELEMENT <name> ( <components> )>
  <more elements>
]>
```

Element Basics

- Defining elements within a DTD is done using an <!ELEMENT> declaration.
  - <!ELEMENT> declarations along with all other declarations within a DTD have no content.
  - <!ELEMENT> declarations are composed of several parts including the element name and the type of information it will contain.
  - The resulting **element names** will be case sensitive.

```
<!ELEMENT   element_name    element_contents>
```

DTD Elements

- The description of an element consists of its name (tag), and a parenthesized description of any nested tags.
  - Includes order of subtags and their multiplicity.
- Leaves (text elements) have #PCDATA in place of nested tags.

What an `<!ELEMENT>` Can Contain

- An `<!ELEMENT>` declaration can contain several different types of content which include the following:
    - `EMPTY`.
    - `PCDATA`.
    - `ANY`.
    - Children Elements

EMPTY

- `<!ELEMENT>` declarations that include the `EMPTY` value allow us to create empty elements within our xml.

- The word `EMPTY` must be entered in uppercase as it is case-sensitive.

```
<!ELEMENT   element_name      EMPTY>
```

PCDATA

- `<!ELEMENT>` declarations that include the value `PCDATA` allow us to include text and other parsable content in our elements within our XML instance file.
  - The word `PCDATA` must be enclosed in parenthesis with a preceding '#' and entered in uppercase as it is case-sensitive.
  - `PCDATA` is text that will be parsed by a parser. Tags inside the text will treated as markup and entities will be expanded.

```
<!ELEMENT    element_name        (#PCDATA)>
```

ANY

- `<!ELEMENT>` declarations that include the value `ANY` allow us include any type of parsable content, including text and other elements, in our elements within our XML instance file.

  - The word `ANY` must be entered in uppercase as it is case-sensitive.

```
<!ELEMENT   element_name      ANY>
```

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: DTD

```
<!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.

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:

```
<!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.

Example (a)

```
<? 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>
    <BAR> …
</BARS>
```

The DTD

The document

Example (b)

- Assume the BARS DTD is in file bar.dtd.

```
<? 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>
```

Get the DTD from the file bar.dtd

Attributes

- Opening tags in XML can have *attributes*, like <A HREF = "…"> in HTML.

- In a DTD,

`<!ATTLIST` <element name>… `>`

gives a list of attributes and their datatypes for this element.

Example: Attributes

- Bars can have an attribute `kind`, which is either sushi, sports, or "other."

```
<!ELEMENT BAR (NAME BEER*)>
  <!ATTLIST BAR kind = "sushi" |
   "sports" | "other">
```

Example: Attribute Use

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

```
<BAR kind = “sushi”>
  <NAME>Akasaka</NAME>
  <BEER><NAME>Sapporo</NAME>
     <PRICE>5.00</PRICE></BEER>
  ...
</BAR>
```

ID’s and IDREF’s

- These are pointers from one object to another, in analogy 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  A = "xyz">
```

Creating IDREF’ s

- To allow objects of type *F* to refer to another object with an ID attribute, give *F* an attribute of type IDREF.

- Or, let the attribute have type IDREFS, so the *F* –object can refer to any number of other objects.

Example: ID's and IDREF's

- Let's redesign our BARS DTD to include both BAR and BEER subelements.
- Both bars and beers will have ID attributes called `name`.
- Bars have PRICE subobjects, 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.

The DTD

```
<!DOCTYPE Bars [
    <!ELEMENT BARS (BAR*, BEER*)>
    <!ELEMENT BAR (PRICE+)>
     <!ATTLIST BAR name = ID>
    <!ELEMENT PRICE (#PCDATA)>
     <!ATTLIST PRICE theBeer = IDREF>
    <!ELEMENT BEER ()>
     <!ATTLIST BEER name = ID, soldBy = IDREFS>
]>
```

Bar objects have name as an ID attribute and have one or more PRICE subobjects.

PRICE objects have a number (the price) and one reference to a beer.

Beer objects have an ID attribute called name, and a soldBy attribute that is a set of Bar names.

Example Document

<BARS>

<BAR name = “JoesBar”>

<PRICE theBeer = “Bud”>2.50</PRICE>

<PRICE theBeer = “Miller”>3.00</PRICE>

</BAR> …

<BEER name = “Bud”, soldBy = “JoesBar, SuesBar,…”>

</BEER> …

</BARS>

QUERYING XML

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.

Principal Kinds of Nodes

1. *Document nodes* represent entire documents.
2. *Elements* are pieces of a document consisting of some opening tag, its matching closing tag (if any), and everything in between.
3. *Attributes* are names that are given values inside opening tags.

Document Nodes

- Formed by doc(URL) or document(URL) (or doc(filename) or document(filename)
- Example: doc("/usr/class/cs4604/bars.xml")
- All XPath (and XQuery) queries refer to a doc node, either explicitly or implicitly.

Example DTD

```
<!DOCTYPE Bars [
  <!ELEMENT BARS (BAR*, BEER*)>
  <!ELEMENT BAR (PRICE+)>
   <!ATTLIST BAR name = ID>
  <!ELEMENT PRICE (#PCDATA)>
   <!ATTLIST PRICE theBeer = IDREF>
  <!ELEMENT BEER ()>
   <!ATTLIST BEER name = ID, soldBy = IDREFS>
]>
```

Example Document

<BARS>

An element node

<BAR name = "JoesBar">

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

<PRICE theBeer = "Gr.Is.">3.00</PRICE>

</BAR> …

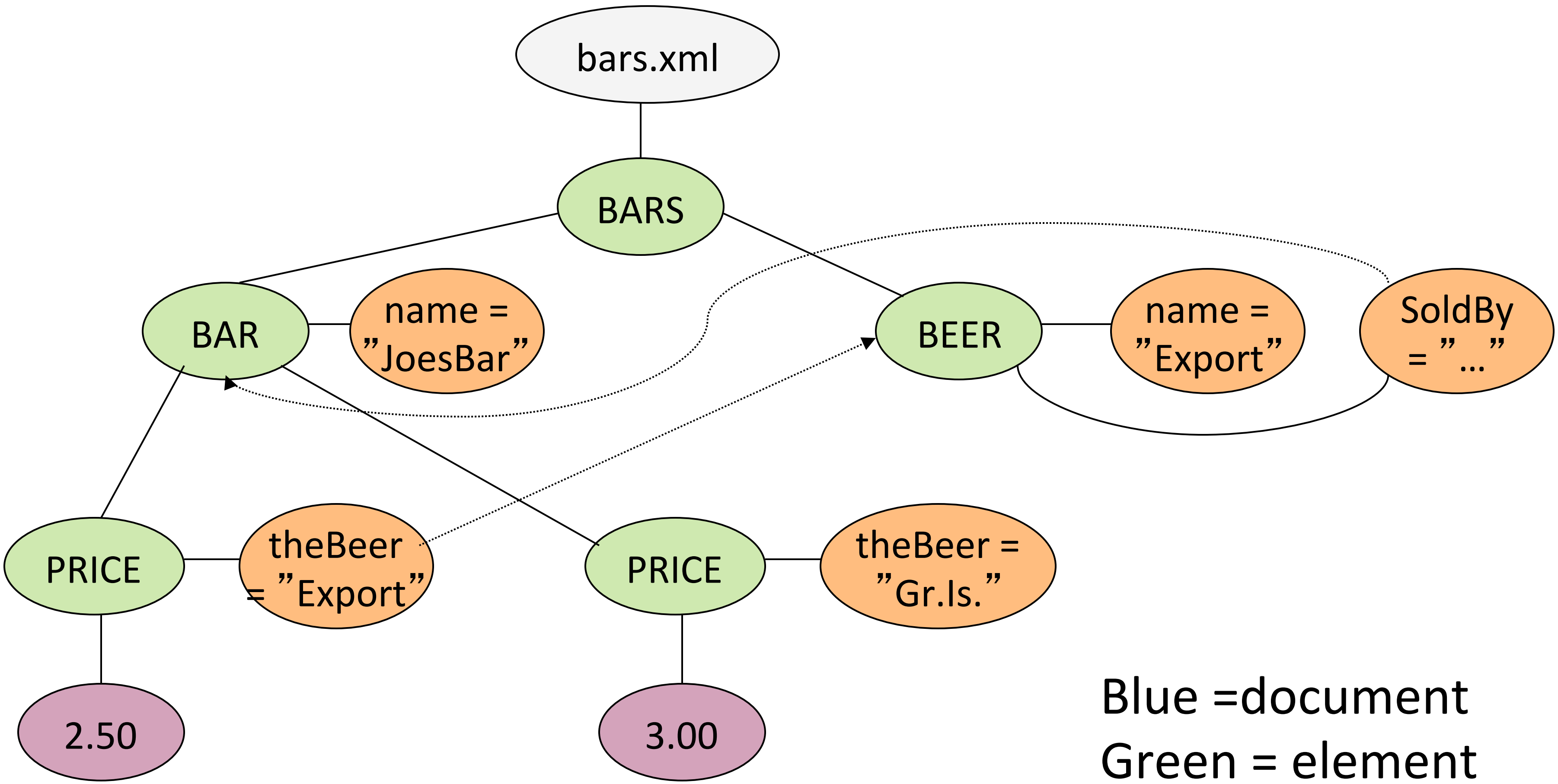
<BEER name = "Export" soldBy = "JoesBar SuesBar … "/> …

An attribute node

</BARS>

Document node is all of this, plus the header (<? xml version…).

Nodes as Semistructured Data



Blue =document
Green = element
Orange = attribute
Purple = primitive value

XPATH and XQUERY

- XPATH is a language for describing paths in XML documents.
  - Really think of the semi-structured data graph and *its* paths.
  - The result of the described path is a sequence of items.
  - Compare with SQL:
    - SQL is a language for describing relations in terms of other relations.
    - The result of a query is a relation (bag) made up of tuples
- XQUERY is a full query language for XML documents with power similar to SQL.

Path Descriptors

- Simple path descriptors are sequences of tags separated by slashes (/).
  - The format used is strongly reminiscent of UNIX naming conventions.
  - Construct the result by starting with just the doc node and processing each tag from the left.
- If the descriptor begins with /, then the path starts at the root and has those tags, in order.
- If the descriptor begins with //, then the path can start anywhere.

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,…">
</BEER> …
</BARS>

/BARS/BAR/PRICE describes the set with these two PRICE objects as well as the PRICE objects for any other bars.

VirginiaTech

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,…”>
</BEER> …
</BARS>

//PRICE describes the same PRICE objects, but only because the DTD forces every PRICE to appear within a BARS and a BAR.

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/*

<BARS>

<BAR name = “JoesBar”>

<PRICE theBeer = “Bud”>2.50</PRICE>

<PRICE theBeer = “Miller”>3.00</PRICE>

</BAR> …

<BEER name = “Bud”, soldBy = “JoesBar, SuesBar,…”>

</BEER> …

</BARS>

/BARS/* captures all BAR and BEER objects, such as these.

Attributes

- In XPATH, we refer to attributes by prepending @ to their name.

- Attributes of a tag may appear in paths as if they were nested within that tag.

Example: /BARS/*/@name

<BARS>

<BAR name = “JoesBar”>

<PRICE theBeer = “Bud”>2.50</PRICE>

<PRICE theBeer = “Miller”>3.00</PRICE>

</BAR> …

<BEER name = “Bud”, soldBy = “JoesBar, SuesBar,…”>

</BEER> …

</BARS>

/BARS/*/@name selects all name attributes of immediate subobjects of the BARS object.

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[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

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 object is selected, along with any other prices for Miller.

Axes

- In general, path expressions allow us to start at the root and execute a sequence of steps to find a set of nodes at each step.
- At each step, we may follow any one of several *axes*.
- The default axis is child:: --- go to any child 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:
  - parent:: = parent(s) of the current node(s).
  - descendant-or-self:: = the current node(s) and all descendants.
    - Note: // is really a shorthand for this axis.
  - ancestor::, ancestor-or-self, etc.

ONLY BRIEFLY (i.e. NOT IN EXAM)

XQuery

- XQuery extends XPath to a query language that has power similar to SQL.
- Uses the same sequence-of-items data model as XPath.
- XQuery is an expression language.
  - Like relational algebra --- any XQuery expression can be an argument of any other XQuery expression.

ONLY BRIEFLY….

FLWR Expressions

- The most important form of XQuery expressions involves for-, let-, where-, return- (FLWR) clauses.
- A query begins with one or more for and/or let clauses.
  - The for's and let's can be interspersed.
- Then an optional where clause.
- A single return clause.

- Form:

  for variable in expression

  let variable := expression

  where condition

  return expression

ONLY BRIEFLY….

Example

- Find all the beer objects where the beer is sold by Joe's Bar for less than 3.00.
- Strategy:

1. $beer will for-loop over all beer objects.
2. For each $beer, let $joe be either the Joe's-Bar object, if Joe sells the beer, or the empty set of bar objects.
3. Test whether $joe sells the beer for < 3.00.

ONLY BRIEFLY....

Example: The Query

Attribute soldBy is of type IDREFS. Follow each ref to a BAR and check if its name is Joe's Bar.

```
FOR $beer IN /BARS/BEER
LET $joe := $beer/@soldBy=>BAR[@name="JoesBar"]
LET $joePrice := $joe/PRICE[@theBeer=$beer/@name]
WHERE $joePrice < 3.00
RETURN <CHEAPBEER>$beer</CHEAPBEER>
```

Only pass the values of $beer, $joe, $joePrice to the RETURN clause if the string inside the PRICE object $joePrice is < 3.00

Find that PRICE subobject of the Joe's Bar object that represents whatever beer is currently $beer.