Chapter 23: XML

Database System Concepts, 6th Ed.

XML

- 23.1 Motivation
- 23.2 Structure of XML Data
- 23.3 XML Document Schema
- 23.4 Querying and Transformation
- 23.5 Application Program Interfaces to XML
- 23.6 Storage of XML Data
- 23.7 XML Applications

XML Introduction [1/2]

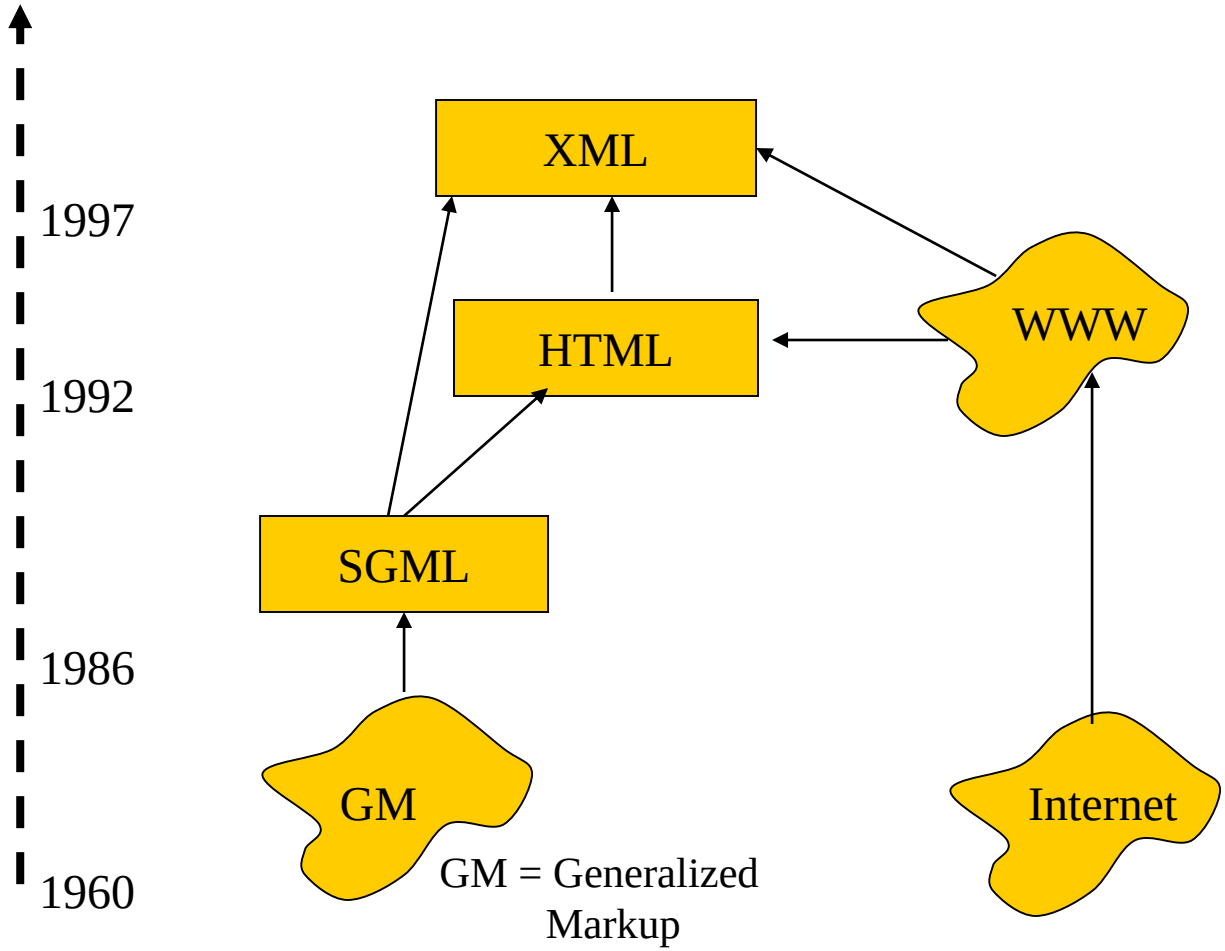
- XML: Extensible Markup Language
  - Defined by the WWW Consortium (W3C)
  - Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML
  - Similar to the old text formatters: Scribe, nroff/troff, Tex/LaTex

- 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

The difference between XML and HTML

- XML is not a replacement for HTML.
- XML and HTML were designed with different goals:
  - XML was designed to transport and store data, with focus on what data is
  - HTML was designed to display data, with focus on how data looks
- XML was created to structure, store, and transport information
- XML language has no predefined tags

History of Tag Languages



XML Introduction [2/2]

- 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 [1/2]

- 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 [2/2]

- 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

- 23.1 Motivation
- 23.2 Structure of XML Data
- 23.3 XML Document Schema
- 23.4 Querying and Transformation
- 23.5 Application Program Interfaces to XML
- 23.6 Storage of XML Data
- 23.7 XML Applications

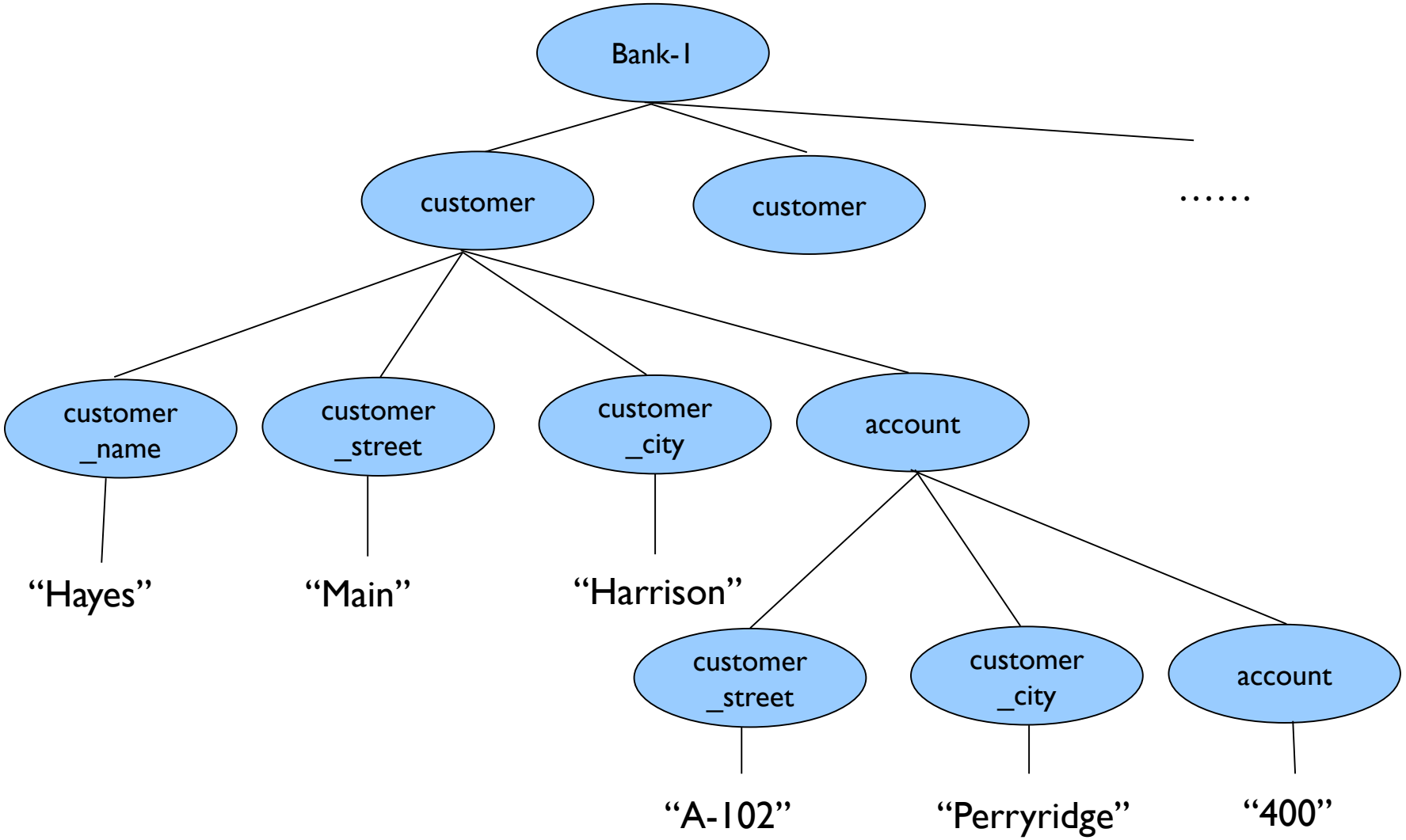
Basic Structure of XML Data

- **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>
  </purchase_order>
```

Example of XML Tree



Motivation for Nesting in XML

- 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

Mixture of Tags and Texts in XML

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

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

  - Useful for document markup, but discouraged for data representation

Attributes in XML Tag

- Elements can have **attributes**

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

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

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

Attributes vs. Subelements in XML

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

XML Namespaces

- XML data has to be exchanged between organizations
  - Same tag names may have different meanings in different organizations, causing confusion on exchanged documents
  - Specifying a unique string as an element name avoids confusion

- Better solution: use unique-name:element-name
- Avoid using long unique names all over document by using XML Namespaces

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

More on XML Syntax

- Elements without subelements or text content can be abbreviated by ending the start tag with a /> and deleting the end tag
  - <course course_id="CS-101" Title="Intro. To Computer Science" dept_name = "Comp. Sci." credits="4" />

- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - <![CDATA[<course> … </course>]]>

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

  CDATA stands for "character data"

XML

- 23.1 Motivation
- 23.2 Structure of XML Data
- 23.3 XML Document Schema
- 23.4 Querying and Transformation
- 23.5 Application Program Interfaces to XML
- 23.6 Storage of XML Data
- 23.7 XML Applications

XML Document Schema

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

Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - All values represented as strings in XML
- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes) >

Element Specification in DTD

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

  <! ELEMENT department (dept_name building, budget)>
  <! ELEMENT dept_name (#PCDATA)>
  <! ELEMENT budget (#PCDATA)>
- Subelement specification may have regular expressions

  <!ELEMENT university ((department | course | instructor | teaches)+)>
  - Notation:
    - "|" - alternatives
    - "+" - 1 or more occurrences
    - "*" - 0 or more occurrences

University DTD

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

Attribute Specification in DTD

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

IDs and IDREFs in DTD

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

University DTD with Attributes

- University DTD with ID and IDREF attribute types.

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

XML data with ID and IDREF attributes: Example

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

Limitations of XML DTDs

- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
  - Order is usually irrelevant in databases (unlike in the document-layout environment from which XML evolved)
  - (A | B)* allows specification of an unordered set, but
    - Cannot ensure that each of A and B occurs only once
- IDs and IDREFs are untyped
  - The *instructors* attribute of an course may contain a reference to another course, which is meaningless
    - *instructors* attribute should ideally be constrained to refer to instructor elements

XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs
- XML Schema supports
  - Typing of values
    - E.g. integer, string, etc
    - Also, constraints on min/max values
  - User-defined, comlex 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.

XML Schema Version of Univ. DTD [1/2]

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

XML Schema Version of Univ. DTD [2/2]

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

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

More features of XML Schema

- Attributes specified by xs:attribute tag:
  - <xs:attribute name = “dept_name”/>
  - adding the attribute use = “required” means value must be specified
- Key constraint: “department names form a key for department elements under the root university element:

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

- Foreign key constraint from course to department:

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

XML Schema vs XML DTD

- XML Schema
  - Created by using xml syntax
  - XML style sheets(XSL) are used with xml schemas
  - Support Namespaces
  - Supports more data types including number and derived data types
  - Easy to create and edit complex content
  - XML schema can be parsable by xml

- XML DTD
  - Not allowed with XSL
  - No namespaces
  - Support only character data types
  - Difficult to reuse
  - Cannot parsable

XML

- 23.1 Motivation
- 23.2 Structure of XML Data
- 23.3 XML Document Schema
- 23.4 Querying and Transformation
- 23.5 Application Program Interfaces to XML
- 23.6 Storage of XML Data
- 23.7 XML Applications

XML Query Language History [1/2]

- Translation of information from one XML schema to another XML schema
- Querying on XML data
- Above two are closely related, and handled by the same tools

- XQuery & XSL
  - Both aim to allow input XML documents to be transformed into XML or other formats
  - Both were developed by separate working groups within W3C, working together to ensure a common approach where appropriate
  - XSLT was primarily conceived as a stylesheet language whose primary goal was to render XML for the human reader
  - XQuery was primarily conceived as a database query language in the tradition of SQL
  - XSLT appeared as a Recommendation in 1999, whereas XQuery is still only a Candidate Recommendation in 2006
  - XSLT is at present much more widely used

XML Query Language History [2/2]

- XPath
  - XPath 1.0: published as a W3C Recommendation in 1999
  - XPath 2.0: in the final stages of the W3C approval process
    - represents a significant increase in the size and capability of the XPath language
    - in fact a subset of XQuery 1.0
- XML-QL
  - A query language for XML
    - handling several issues that the XML standard does not address
  - Just a submission to the W3c in 1998 & no later advance
- SQL/XML is an extension of SQL that is part of ANSI/ISO SQL 2003

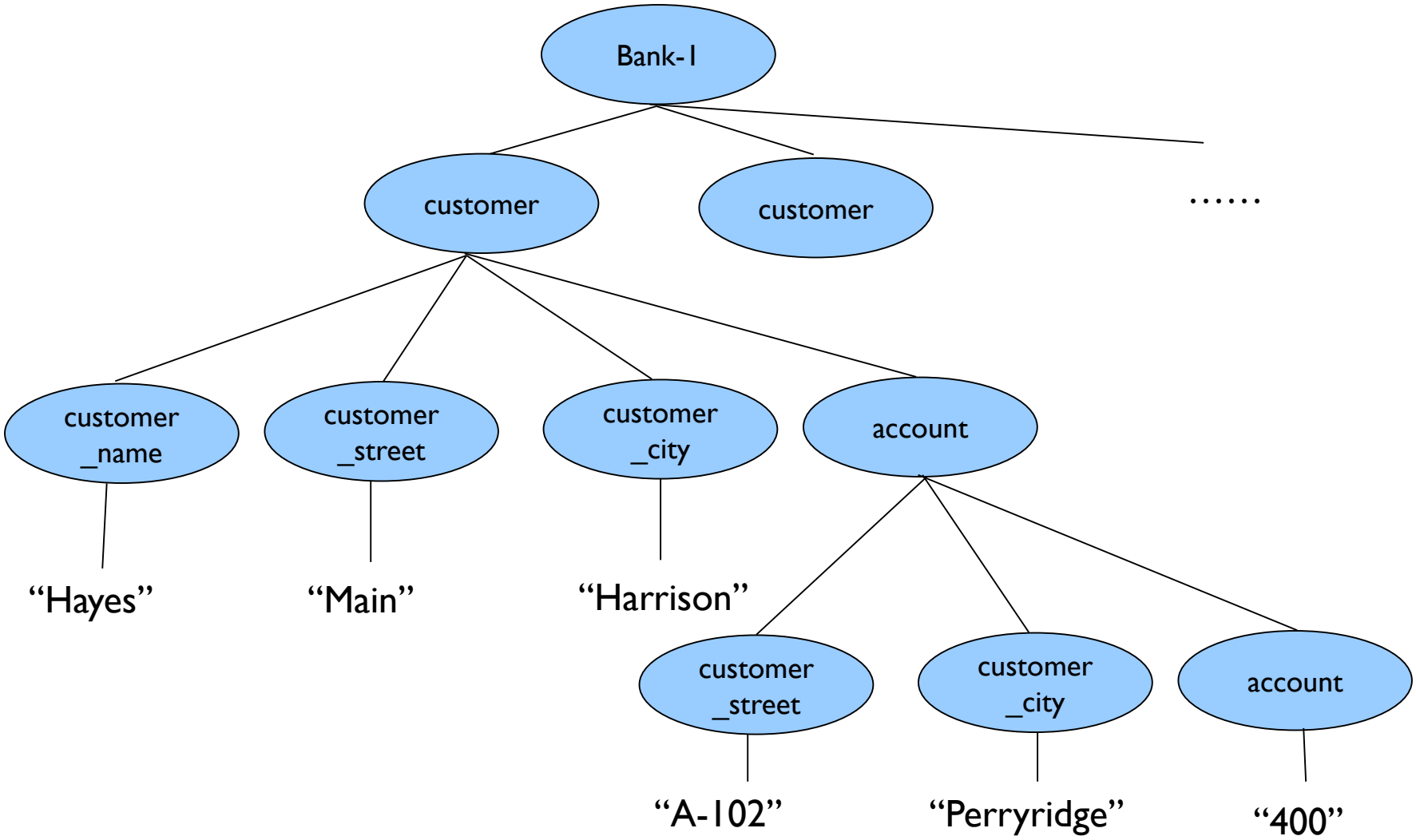
Querying and Transforming XML Data

- 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

Example of XML Tree



XPath [1/2]

- 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

XPath [2/2]

- The initial "/" denotes root of the document (above the top-level tag)
- Path expressions are evaluated left to right
  - Each step operates on the set of instances produced by the previous step
- Selection predicates may follow any step in a path, in [ ]
  - E.g. /university-3/course[credits >= 4]
    - returns account elements with a balance value greater than 400
    - /university-3/course[credits] returns account elements containing a credits subelement
- Attributes are accessed using "@"
  - E.g. /university-3/course[credits >= 4]/@course_id
    - returns the course identifiers of courses with credits >= 4
  - IDREF attributes are not dereferenced automatically (more on this later)

Functions in XPath

- XPath provides several functions
  - The function count() at the end of a path counts the number of elements in the set generated by the path
    - E.g. /university-2/instructor[count(./teaches/course)> 2]
      - Returns instructors teaching more than 2 courses (on university-2 schema)
  - Also function for testing position (1, 2, ..) of node w.r.t. siblings
- Boolean connectives and and or and function not() can be used in predicates
- IDREFs can be referenced using function id()
  - id() can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
  - E.g. /university-3/course/id(@dept_name)
    - returns all department elements referred to from the dept_name attribute of course elements.

More XPath Features

- Operator "|" used to implement union
  - E.g. /university-3/course[@dept name="Comp. Sci"] | /university-3/course[@dept name="Biology"]
    - Gives union of Comp. Sci. and Biology courses
    - However, "|" cannot be nested inside other operators.
- "//" can be used to skip multiple levels of nodes
  - E.g. /university-3//name
    - finds any name element *anywhere* under the /university-3 element, regardless of the element in which it is contained.
- A step in the path can go to parents, siblings, ancestors and descendants of the nodes generated by the previous step, not just to the children
  - "//", described above, is a short from for specifying "all descendants"
  - ".." specifies the parent.
- doc(name) returns the root of a named document

XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
  - The textbook description is based on a January 2005 draft of the standard. The final version may differ, but major features likely to stay unchanged.
- 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

FLWOR Syntax in XQuery

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- Simple FLWOR expression in XQuery
  - find all courses with credits > 3, with each result enclosed in an <course_id> .. </course_id> tag

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

  - Items in the **return** clause are XML text unless enclosed in {}, in which case they are evaluated
- Let clause not really needed in this query, and selection can be done In XPath. Query can be written as:

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

- Alternative notation for constructing elements:

```
return element course_id { element  $x/@course_id }
```

Joins in XQuery

- Joins are specified in a manner very similar to SQL

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

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

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

Nested Queries in XQuery

- The following query converts data from the flat structure for university information into the nested structure used in university-1

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

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

Grouping and Aggregation in XQuery

- Nested queries are used for grouping

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

Sorting in XQuery

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

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

- Use **order by** $i/name **descending** to sort in descending order
- Can sort at multiple levels of nesting (sort departments by dept_name, and by courses sorted to course_id within each department)

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

Functions and Other XQuery Features

- User defined functions with the type system of XMLSchema

```
declare function local:dept_courses($iid as xs:string)
        as element(course)*
{
    for $i in /university/instructor[IID = $iid],
        $c in /university/courses[dept_name = $i/dept name]
    return $c
}
```

- Types are optional for function parameters and return values
- The * (as in decimal*) indicates a sequence of values of that type
- Universal and existential quantification in where clause predicates
  - **some** $e **in** *path* **satisfies** *P*
  - **every** $e **in** *path* **satisfies** *P*
  - Add **and fn:exists($e)** to prevent empty $e from satisfying **every** clause
- XQuery also supports If-then-else clauses

XML Stylesheet Language (XSLT)

- A **stylesheet** stores formatting options for a document, usually separately from document
  - E.g. an HTML style sheet may specify font colors and sizes for headings, etc.
- The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language
  - Can translate XML to XML, and XML to HTML
- XSLT transformations are expressed using rules called **templates**
  - Templates combine selection using XPath with construction of results

XML

- 23.1 Motivation
- 23.2 Structure of XML Data
- 23.3 XML Document Schema
- 23.4 Querying and Transformation
- 23.5 Application Program Interfaces to XML
- 23.6 Storage of XML Data
- 23.7 XML Applications

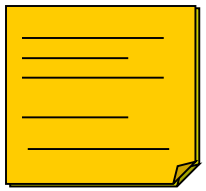
XML Application Program Interface

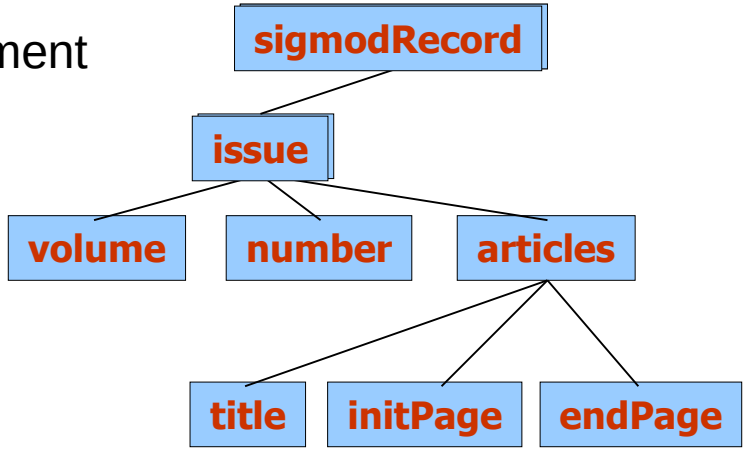
- There are two standard application program interfaces to XML data:
  - **SAX** (Simple API for XML)
    - Based on parser model, user provides event handlers for parsing events
      - E.g. start of element, end of element
  - **DOM** (Document Object Model)
    - **XML** data is parsed into a tree 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

DOM API

- Characteristics
  - Hierarchical (tree) object model for XML documents
  - Associate list of children with every node
  - Preserves the sequence of the elements in the XML documents
- DOM interfaces
  - *Node* : The base data type of the DOM.
  - *Element* : The vast majority of the objects you'll deal with are Elements.
  - *Attr* : Represents an attribute of an element.
  - *Text* : The actual content of an Element or Attr.
  - *Document* : Represents the entire XML document

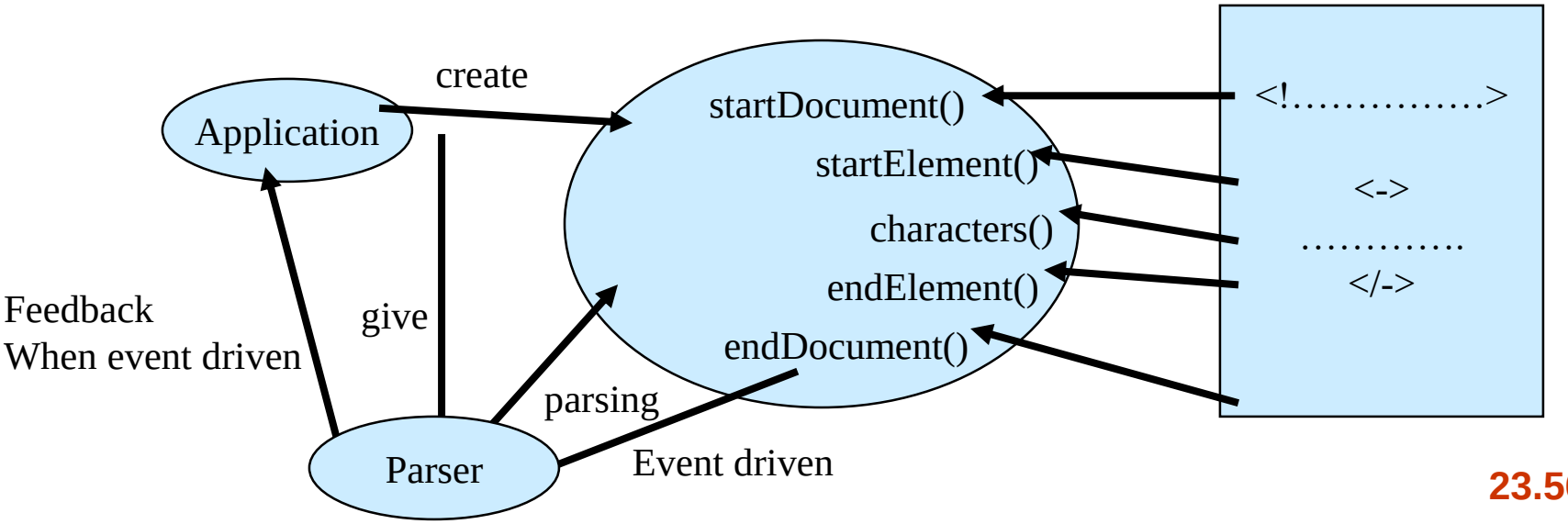
XML document





SAX API

- DOM : expensive to materialize for a large XML collection
- Characteristics
  - Event-driven : fire an event for every open tag/end tag
  - Does not require full parsing
  - Enables custom object model building
- The SAX API actually defines four interfaces for handling events
  - *EntityHandler : TDHandler : DocumentHandler :ErrorHandler*
- All of these interfaces are implemented by *HandlerBase*.



DOM Parsing vs SAX Parsing

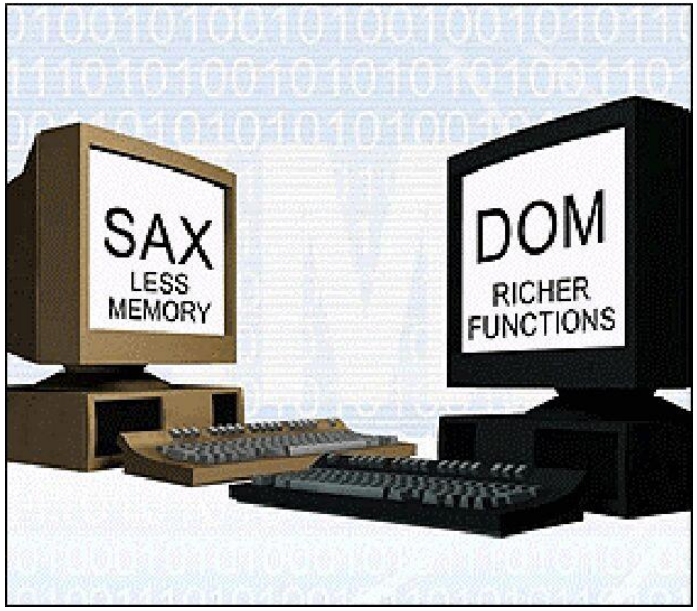
```
<?XML version="1.0">
<Book>
   <Author>Sonny</Author>
   <Title>XML Bible</Title>
   <Price>13000</Price>
</Book>
```

DOM Parsing

SAX Parsing

startElement : **Book**
startElement : Author
characters : Sonny
endElement : Author
startElement : Title
characters : XML Bible
endElement : Title
startElement : Price
characters : 13000
endElement : Price
endElement : **Book**

DOM vs SAX [1/3]



- Why use DOM?
  - Need to know a lot about the structure of a document
  - Need to move parts of the document around
  - Need to use the information in the document more than once
- Why use SAX?
  - Only need to extract a few elements from an XML document

DOM vs SAX [2/3]

<book id="1">
<verse>
Sing, O goddess, the anger of Achilles son of Peleus, that brought countless ills upon the Achaeans. Many a brave soul did it send hurrying down to Hades, and many a hero did it yield a prey to dogs and vultures, for so were the counsels of Jove fulfilled from the day on which the son of
Atreus, king of men, and great Achilles, first fell out with one another.
</verse>
<verse>
And which of the gods was it that set them on to quarrel? It was the son of Jove and Leto; for he was angry with the king and sent a pestilence upon
...

- Doing this with the DOM would take a lot of memory
  SAX API would be much more efficient

DOM vs SAX [3/3]

```
…
<address>
      <name><first-name>Mary</first-name><last-name>McGoon</last-name></name>
      <street>1401 Main Street</street> <city>Anytown</city>
<state>NC</state><zip>34829</zip>
</address>
<address>
     <name>…
     <street> …
</address>
```

If we were parsing an XML document containing 10,000 address, and we wanted to sort them by last name??
DOM would automatically store all of the data
We could use DOM functions to move the nodes n the DOM tree

XML

- 23.1 Motivation
- 23.2 Structure of XML Data
- 23.3 XML Document Schema
- 23.4 Querying and Transformation
- 23.5 Application Program Interfaces to XML
- 23.6 Storage of XML Data
- 23.7 XML Applications

Storage of XML Data

- Non-relational data stores
  - Flat files : Natural for storing XML
    - But has all problems discussed in Chapter 1 (no concurrency, no recovery,…)
  - XML database
    - Database built specifically for storing XML data, supporting DOM model and declarative querying
    - Currently no commercial-grade systems
- Relational databases
  - Data must be translated into relational form
  - Advantage: mature database systems
  - Disadvantages: overhead of translating data and queries
- Alternatives:
    - String Representation
    - Tree Representation
    - Map to relations

String Representation of XML Data [1/2]

- Store each top level element as a string field of a tuple in a relational database
  - Use a single relation to store all elements, or
  - Use a separate relation for each top-level element type
    - E.g. account, customer, depositor relations
      - Each with a string-valued attribute to store the element
- Indexing:
  - Store values of subelements/attributes to be indexed as extra fields of the relation, and build indices on these fields
    - E.g. customer_name or account_number
  - Some database systems support **function indices,** which use the result of a function as the key value.
    - The function should return the value of the required subelement/attribute

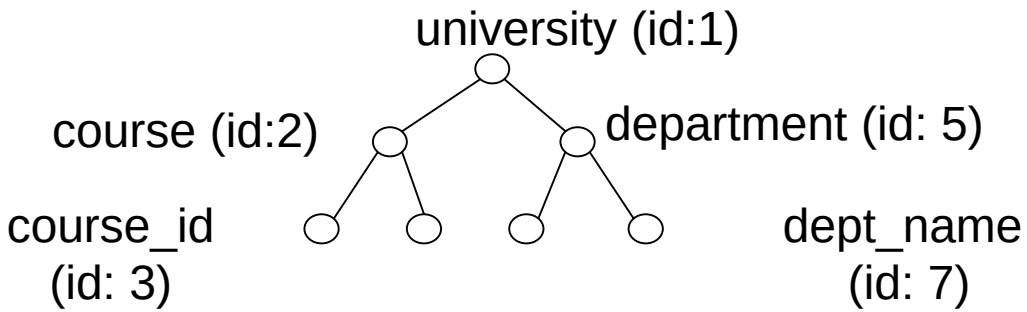
String Representation of XML Data [2/2]

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

Tree Representation of XML Data

- **Tree representation:** model XML data as tree and store using relations

nodes(id, parent_id, type, label, value)



- Each element/attribute is given a unique identifier
- Type indicates element/attribute
- Label specifies the tag name of the element/name of attribute
- Value is the text value of the element/attribute
- Can add an extra attribute *position* to record ordering of children
- Benefit: Can store any XML data, even without DTD
- Drawbacks:
  - Data is broken up into too many pieces, increasing space overheads
  - Even simple queries require a large number of joins, which can be slow

Mapping XML Data to Relations

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

Sample XML Data File

보조자료

```
<bank>
    <account>
        <account-number>A101</account-number>
        <branch-name>Downtown</branch-name>
        <balance>500</balance>
    </account>
    <account>
        <account-number>A102</account-number>
        <branch-name>Perryridge</branch-name>
        <balance>400</balance>
    </account>
    <account>
        <account-number>A201</account-number>
        <branch-name>Brighton</branch-name>
        <balance>900</balance>
    </account>
```

continued…

```
    <customer>
        <customer-name>Johnson</customer-name>
        <customer-street>Alma</customer-street>
        <customer-city>Palo Alto</customer-city>
    </customer>
    <customer>
        <customer-name>Hayes</customer-name>
        <customer-street>Main</customer-street>
        <customer-city>Harrison</customer-city>
        </customer>
    <depositor>
        <account-number>A-101</account-number>
        <customer-name>Johnson</customer-name>
    </depositor>
    <depositor>
        <account-number>A-201</account-number>
        <customer-name>Johnson</customer-name>
    </depositor>
    <depositor>
        <account-number>A-102</account-number>
        <customer-name>Hayes</customer-name>
    </depositor>
    </bank>
```

보조자료

Store as a String – 1st Method [1/2]

- Store as string – The 1st method
  - 각 child element들의 top-level element를 individual tuple (string)으로 저장
  - 앞 페이지의 XML은 *elements(data)* relation의 tuple들로 저장 가능
  - Attribute data는 각 XML elements (**account**, **customer**, or **depositor**)들을 string 형식으로 저장
- 장점
  - 개념과 구현이 간단하다
- 단점
  - Database system은 저장된 element들의 스키마를 알지 못함
  - 모든 **account** element들을 찾거나 account number A-401을 가진 **account** element를 찾는 단순한 query도 relation의 전체를 스캔하여야 함

Store as a String – 1st Method [2/2]

보조자료

❖ *elements*(*data*)

DATA
<account> <account-number> A101 </account-number> <branch-name> Downtown </branch-name> <balance> 500 </balance> </account>
<account> <account-number> A102 </account-number> <branch-name> Perryridge </branch-name> <balance> 400 </balance> </account>
<account> <account-number> A201 </account-number> <branch-name> Brighton </branch-name> <balance> 900 </balance> </account>
<customer> <customer-name> Johnson </customer-name> <customer-street> Alma </customer-street> <customer-city> Palo Alto </customer-city> </customer>
<customer> <customer-name> Hayes </customer-name> <customer-street> Main </customer-street> <customer-city> Harrison </customer-city> </customer>
<depositor> <account-number> A-101 </account-number> <customer-name> Johnson </customer-name> </depositor>
<depositor> <account-number> A-201 </account-number> <customer-name> Johnson </customer-name> </depositor>
<depositor> <account-number> A-102 </account-number> <customer-name> Hayes </customer-name> </depositor>

※ 줄바꿈 문자는 생략된 형태로 표현

Store as a String – 2nd Method [1/4]

보조자료

- Store as string – The 2nd method
  - 첫번째 방법의 문제점을 부분적으로 해결하기 위한 방법
  - 각 element들을 타입 별로 다른 relation에 저장
  - 중요한 element들을 relation의 attribute들로 저장 → indexing을 가능케 함
  - *elements(data)* 스키마의 변경
    - data라는 attribute를 가진 account-elements, customer-elements, depositor-elements relation들로 저장
    - 각 relation들은 account-number, customer-name처럼 일부 중요한 subelement들의 값을 저장
    - 특정 account number를 갖는 account를 찾는 쿼리를 효율적으로 처리 가능
- 장점
  - 첫번째 방법에 비해 일부 query를 효율적으로 처리할 수 있다
- 단점
  - DTD와 같은 XML data의 타입 정보에 의존적

Store as a String – 2nd Method [2/4]

보조자료

❖***account-elements***(*data, account_number*)

DATA	ACCOUNT_NUMBER
<account> <account-number> A-101 </account-number><branch-name> Downtown </branch-name> <balance> 500</balance> </account>	A101
<account> <account-number> A-102 </account-number> <branch-name> Perryridge </branch-name> <balance> 400</balance> </account>	A102
<account> <account-number> A-201 </account-number> <branch-name> Brighton </branch-name> <balance> 900</balance> </account>	A201

- **account** element를 저장하기 위한 테이블
- data attribute에 해당 element를 string으로 저장
- Sub element *account_number* 는 중요하므로 별도의 attribute로 저장

Store as a String – 2nd Method [3/4]

보조자료

❖***customer-elements***(*data, customer_name*)

DATA	CUSTOMER_NAME
<customer> <customer-name> Johnson </customer-name> <customer-street> Alma </customer-street> <customer-city> Palo Alto </customer-city> </customer>	Johnson
<customer> <customer-name> Hayes </customer-name> <customer-street> Main </customer-street> <customer-city> Harrison </customer-city> </customer>	Hayes

- **customer** element를 저장하기 위한 테이블
- data attribute에 해당 element를 string으로 저장
- Sub element *customer_name*은 중요하므로 별도의 attribute로 저장

보조자료

Store as a String – 2nd Method [4/4]

❖***depositor-elements***(*data, account_number, customer_name*)

DATA	ACCOUNT_NUMBER	CUSTOMER_NAME
<depositor> <account-number> A-101 </account-number> <customer-name> Johnson </customer-name> </depositor>	A-101	Johnson
<depositor> <account-number> A-201 </account-number> <customer-name>Johnson</customer-name> </depositor>	A-201	Johnson
<depositor> <account-number> A-102 </account-number> <customer-name> Hayes </customer-name> </depositor>	A-102	Hayes

- **depositor** element를 저장하기 위한 테이블
- data attribute에 해당 element를 string으로 저장
- Sub element *account_number*와 *customer_name*은 중요하므로 별도의 attribute로 저장

Tree Representation [1/4]

보조자료

- 방법
  - 임의의 XML Data를 tree 형태로 표현
    - Relation nodes와 child로 각 XML의 element와 attribute를 표시하고 parent와 child의 관계를 표시
  - nodes (id, type, label, value)
  - child (child-id, parent-id)
- 장점
  - XML의 정보가 relation 형태로 직접 표현할 수 있다. (효율적 저장능력)
  - 여러 XML 질의들이 SQL로 변환될 수 있다.
  - 변환된 질의가 DB 시스템내에서 실행가능하다.
- 단점
  - XML의 각 요소들이 많은 조각들로 나누어진다.
  - 그러한 요소들을 다시 원상태로 하려면 많은 join이 필요
  - DTD를 알아야 한다

보조자료

Tree Representation [2/4]

- nodes (table)
  - XML의 각 element와 attribute 정보를 나타낸다.
  - Relation의 각 필드는 id, type, label, value 이다.
  - null 값은 ‘null’로, character type은 ‘char’ 로 Integer type은 integer로 표현

id	type	label	value
B0	char	bank	null
A1	char	account	null
A2	char	account	null
A3	char	account	null
C1	char	customer	null
C2	char	customer	null
D1	char	depositor	null
D2	char	depositor	null
D3	char	depositor	null

Tree Representation [3/4]

보조자료

- 사례 : **nodes**

A1ac1	char	account-number	A-101
A1br1	char	branch-name	Downtown
A1ba1	integer	balance	500
A2ac2	char	account-number	A-102
A2br2	char	branch-name	Perryridge
A2ba2	integer	balance	400
A3ac3	char	account-number	A-201
A3br3	char	branch-name	Brighton
A3ba3	integer	balance	900
C1cn1	char	customer-name	Johnson
C1cs1	char	customer-street	Alma
C1cc1	char	customer-city	Palo Alto

C2cn2	char	customer-name	Hayes
C2cs2	char	customer-street	Main
C2cc2	char	customer-city	Harrison
D1ac1	char	account-number	A-101
D1cn1	char	customer-name	Johnson
D2ac2	char	account-number	A-201
D2cn2	char	customer-name	Johnson
D3ac3	char	account-number	A-102
D3cn3	char	customer-name	Hayes

보조자료

Tree Representation [4/4]

- child (table)
  - XML의 각 요소의 상,하 관계를 나타낸다.
  - Relation의 각 필드는child-id, parent-id 이다.
  - 각 필드 type은 character type으로 정한다

child-id	parent-id
A1	B0
A2	B0
A3	B0
C1	B0
C2	B0
D1	B0
D2	B0
D3	B0
A1	B0

A1ac1	A1
A1br1	A1
A1ba1	A1
A2ac2	A2
A2br2	A2
A2ba2	A2
A3ac3	A3
A3br3	A3
A3ba3	A3
C1cn1	C1
C1cs1	C2
C1cc1	C3

C2cn2	C2
C2cs2	C2
C2cc2	C2
D1ac1	D1
D1cn1	D1
D2ac2	D2
D2cn2	D2
D3ac3	D3
D3cn3	D3

보조자료

Map to Relation [1/2]

- DTD

```
<!DOCTYPE bank[
        <!ELEMENT bank ((account-customer-depositor)+)>
        <!ELEMENT account (acount-number branch-name balance)>
        <!ELEMENT customer (customer-name customer-street
customer-city)>
        <!ELEMENT depositor (customer-name account-number)>
        <!ELEMENT account-number (#PCDATA)>
        <!ELEMENT branch-name (#PCDATA)>
        <!ELEMETN balance (#PCDATA)>
        <!ELEMENT customer-name (#PCDATA)>
        <!ELEMENT customer-street (#PCDATA)>
        <!ELEMENT customer-city (#PCDATA)>
]>
```

Map to Relation [2/2]

보조자료

- Account Relation

Account-number	Branch-name	Balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900

- Customer Relation

customer-name	Customer-street	Customer-city
Jonhnson	Alma	Palo Alto
Hayes	Main	Harrison

- Depositor Relation

Customer-name	Account-number
Johnson	A-101
Johnson	A-201
Hayes	A-102

Storing XML Data in Relational Systems

- Applying above ideas to department elements in university-1 schema, with nested course elements, we get
  department(*id*, *dept_name*, *building*, *budget*)
  course(*parent id*, *course_id*, *dept_name*, *title*, *credits*)
- **Publishing**: process of converting relational data to an XML format
- **Shredding**: process of converting an XML document into a set of tuples to be inserted into one or more relations
- XML-enabled database systems support automated publishing and shredding
- Many systems offer *native storage* of XML data using the **xml** data type
- Special internal data structures and indices are used for efficiency

XML Supports in Relational Systems

- *Publishing and Shredddding*
  - *Publishing*: process of converting relational data to an XML format
  - *Shredding*: process of converting an XML document into a set of tuples to be inserted into one or more relations
  - XML-enabled database systems support automated publishing and shredding

- Some systems offer *native storage* of XML data using the **xml** data type
  - Attribute type: XML
  - Underlying implementation is based on CLOB and BLOB
  - Special internal data structures and indices are used for efficiency
  - Xpath and Xquery are supported

SQL/XML Standard

- XQuery & SQL/XML are 2 standards that use declarative, portable queries to return XML by querying data
  - XQuery is XML-centric, while SQL/XML is SQL-centric
  - For a SQL programmer, SQL/XML easy to learn because it involves only a few small additions to the existing SQL language
    - Cf. SQL/XML is completely different from Microsoft's SQLXML, a proprietary technology used in SQL Server
- A standard extension of SQL allowing the creation of nested XML output (publishing)
  - Mapping SQL types into XML schema types
  - Map relational schemas to XML

SQL/XML Tuple Creation

- Each output tuple is mapped to an XML element *row*

```
<bank>
  <account>
    <row>
      <account_number> A-101      </account_number>
      <branch_name>    Downtown </branch_name>
      <balance>          500        </balance>
    </row>
      … more rows if there are more output tuples…
      </account>
</bank>
```

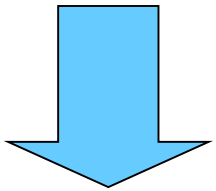
SQL/XML Publishing Fuctions

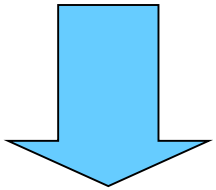
xmlelement()	Creates an XML element, allowing the name to be specified
xmlattributes()	Creates XML attributes from columns, using the name of each column as the name of the corresponding attribute
xmlroot()	Creates the root node of an XML document
xmlcomment()	Creates an XML comment
xmlpi()	Creates an XML processing instruction
xmlparse()	Parses a string as XML and returns the resulting XML structure
xmlforest()	Creates XML elements from columns, using the name of each column as the name of the corresponding element
xmlconcat()	Combines a list of individual XML values to create a single value containing an XML forest
xmlagg()	Combines a collection of rows, each containing a single XML value, to create a single value containing an XML forest

SQL/XML Example [1/2]

```
select c.CustId, c.Name as CustName
from customers c
```

```
CustId  CustName
------- ------------------
1       Woodworks
4       Hardware Shop
6       Photo Shop
8       Computer Supplies
```





```
select xmlelement(name "Customer",
       xmlelement(name "CustId", c.CustId),
       xmlelement(name "CustName", c.Name)
       xmlelement(name "City", c.City))
from Customers  c
```

```
<Customer>
  <CustId>1</CustId>
  <CustName>Woodworks</CustName>
  <City>Baltimore</City>
</Customer>
```

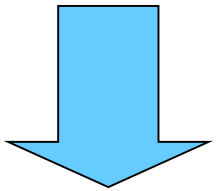
```
select xmlelement(name "Customer",
      xmlforest(c.CustId, c.Name as CustName, c.City))
from Customers  c
```

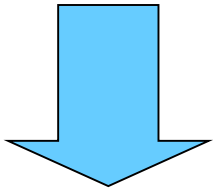
SQL/XML Example [2/2]

```
select *
from Customers c, Projects p
where c.CustId = p.CustId
order by c.CustId, p.ProjId
```

Customer relation

Projects relation





```
select xmlelement(name "CustomerProj",
   xmlforest(c.CustId, c.Name as CustName, p.ProjId, p.Name as ProjName))
from Customers c, Projects p
where p.CustId=c.CustId
order by c.CustId
```

```
<CustomerProj>
  <CustId>1</CustId>
  <CustName>Woodworks</CustName>
  <ProjId>1</ProjId>
  <ProjName>Medusa</ProjName>
</CustomerProj>
<CustomerProj>
  <CustId>4</CustId>
  <CustName>Hardware Shop</CustName>
  <ProjId>2</ProjId>
  <ProjName>Pegasus</ProjName>
</CustomerProj>
<CustomerProj>
  <CustId>4</CustId>
  <CustName>Hardware Shop</CustName>
  <ProjId>8</ProjId>
  <ProjName>Typhon</ProjName>
</CustomerProj>
```

SQL Extensions in SQL/XML

- **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

- 23.1 Motivation
- 23.2 Structure of XML Data
- 23.3 XML Document Schema
- 23.4 Querying and Transformation
- 23.5 Application Program Interfaces to XML
- 23.6 Storage of XML Data
- 23.7 XML Applications

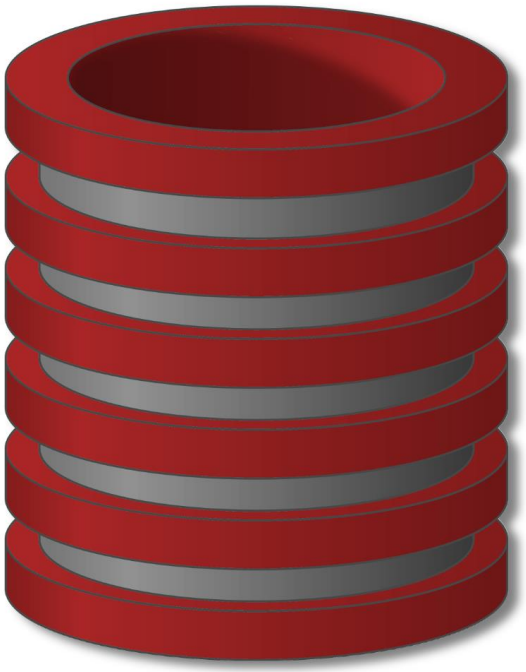
XML Applications

- 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 Service

Web Services

- The Simple Object Access Protocol (SOAP) standard:
  - Invoking 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

By Professor Jennifer Widom at Stanford University



JSON Data

Introduction

JavaScript Object Notation (JSON)

- Standard for “serializing” data objects, usually in files
- Human-readable, useful for data interchange
- Also useful for representing & storing semistructured data

```
{ "Books":
  [
    { "ISBN":"ISBN-0-13-713526-2",
      "Price":85,
      "Edition":3,
      "Title":"A First Course in Database Systems",
      "Authors":[ {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
    ,
    { "ISBN":"ISBN-0-13-815504-6",
      "Price":100,
      "Remark":"Buy this book bundled with 'A First Course' - a great deal!",
      "Title":"Database Systems:The Complete Book",
      "Authors":[ {"First_Name":"Hector", "Last_Name":"Garcia-Molina"},
                  {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
  ]
```

JavaScript Object Notation (JSON)

- No longer tied to JavaScript
- Parsers for many languages

```
{ "Books":
  [
    { "ISBN":"ISBN-0-13-713526-2",
      "Price":85,
      "Edition":3,
      "Title":"A First Course in Database Systems",
      "Authors":[ {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
    ,
    { "ISBN":"ISBN-0-13-815504-6",
      "Price":100,
      "Remark":"Buy this book bundled with 'A First Course' - a great deal!",
      "Title":"Database Systems:The Complete Book",
      "Authors":[ {"First_Name":"Hector", "Last_Name":"Garcia-Molina"},
                  {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
  ]
```

```
{ "Books":
  [
    { "ISBN":"ISBN-0-13-713526-2",
      "Price":85,
      "Edition":3,
      "Title":"A First Course in Database Systems",
      "Authors":[ {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
    ,
    { "ISBN":"ISBN-0-13-815504-6",
      "Price":100,
      "Remark":"Buy this book bundled with 'A First Course' - a great deal!",
      "Title":"Database Systems:The Complete Book",
      "Authors":[ {"First_Name":"Hector", "Last_Name":"Garcia-Molina"},
                  {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
  ],
  "Magazines":
  [
    { "Title":"National Geographic",
      "Month":"January",
      "Year":2009 }
    ,
    { "Title":"Newsweek",
      "Month":"February",
      "Year":2009 }
  ]
}
```

Basic constructs (recursive)

- Base values
  number, string, boolean, …
- Objects { }
  sets of label-value pairs
- Arrays [ ]
  lists of values

Relational Model vs JSON

	Relational	JSON
Structure		
Schema		
Queries		
Ordering		
Implementation		

Relational Model versus JSON

By Prof J. Widom

	Relational	JSON
Structure	Tables	Nested Sets Arrays
Schema	Fixed in advance	"self-describing" Flexible
Queries	Simple expressive languages	∅ widely used
Ordering	None.	Arrays.
Implementation	Native systems.	Coupled with PLs. NoSQL Systems.

XML vs JSON

By Prof J. Widom

	XML	JSON
Verbosity		
Complexity		
Validity		
Prog. Interface		
Querying		

XML versus JSON

By Prof J. Widom

	XML	JSON
Verbosity	More	Less
Complexity	More	Less
Validity	DTDs widely used XSDs	JSON Schema not widely used
Prog. Interface	Clunky "Impedence mismatch"	More direct
Querying	XPath - XQuery XSLT -	JSON Path JAQL JSON Query

Syntactically valid JSON

Adheres to basic structural requirements

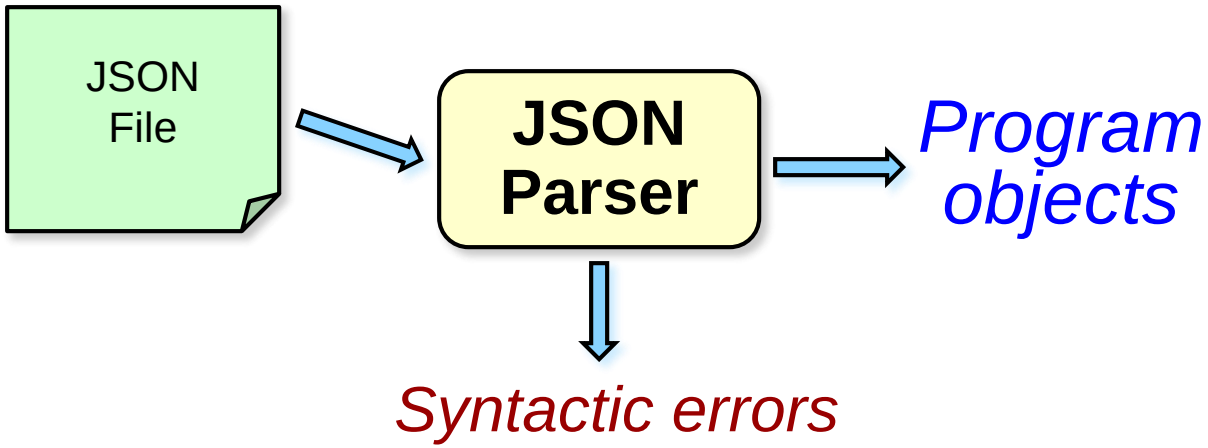
- Sets of label-value pairs
- Arrays of values
- Base values from predefined types

```
{ "Books":
  [
    { "ISBN":"ISBN-0-13-713526-2",
      "Price":85,
      "Edition":3,
      "Title":"A First Course in Database Systems",
      "Authors":[ {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
    ,
    { "ISBN":"ISBN-0-13-815504-6",
      "Price":100,
      "Remark":"Buy this book bundled with 'A First Course' - a great deal!",
      "Title":"Database Systems:The Complete Book",
      "Authors":[ {"First_Name":"Hector", "Last_Name":"Garcia-Molina"},
                  {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
  ]
```

Syntactically valid JSON

Adheres to basic structural requirements

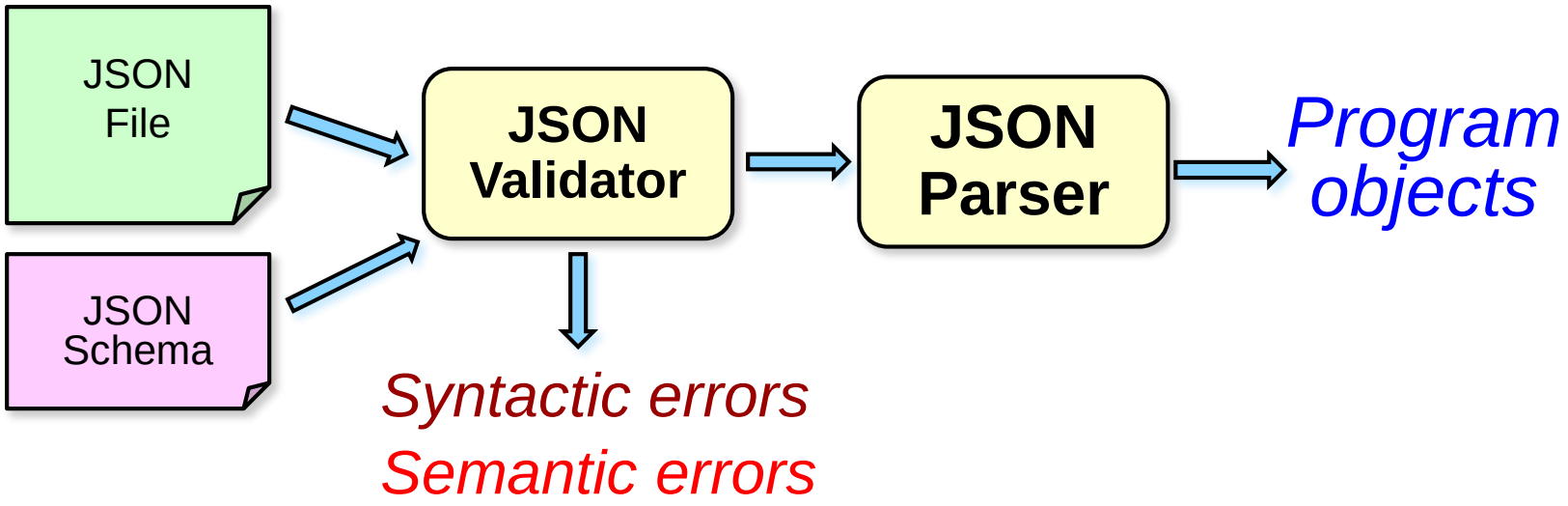
- Sets of label-value pairs
- Arrays of values
- Base values from predefined types



Semantically valid JSON

Adheres to basic structural requirements

+ conforms to specified schema



JavaScript Object Notation (JSON)

- Standard for “serializing” data objects in human-readable format
- Useful for data interchange, and for representing & storing semistructured data

```
{ "Books":
  [
    { "ISBN":"ISBN-0-13-713526-2",
      "Price":85,
      "Edition":3,
      "Title":"A First Course in Database Systems",
      "Authors":[ {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
    ,
    { "ISBN":"ISBN-0-13-815504-6",
      "Price":100,
      "Remark":"Buy this book bundled with 'A First Course' - a great deal!",
      "Title":"Database Systems:The Complete Book",
      "Authors":[ {"First_Name":"Hector", "Last_Name":"Garcia-Molina"},
                  {"First_Name":"Jeffrey", "Last_Name":"Ullman"},
                  {"First_Name":"Jennifer", "Last_Name":"Widom"} ] }
  ]
```

(Example) JSON Parsing with JAVA

- Used json_simple library
  - String jsonInfo

```
[
    {"userId" : "aaa", "email" : "aaa@gmail.com", "age" : 22, "gender" : "male"},
    {"userId" : "bbb", "email" : "bbb@gmail.com", "age" : 31, "gender" : "female"},
    {"userId" : "ccc", "email" : "ccc@gmail.com", "age" : 25, "gender" : "male"},
]
```

  - Parsing Code

```
JSONArray jsonArray = (JSONArray) JSONValue.parse(jsonInfo);

JSONObject jsonObject1 = (JSONObject) jsonArray.get(1);
System.out.println(jsonObject1.get("email"));    /// bbb@gmail.com

JSONObject jsonObject2 = (JSONObject) jsonArray.get(2);
System.out.println(jsonObject2.get("userId"));   /// ccc
```

추가자료

What about XML?

- XML query languages like XML-QL choose graph nodes to operate on via *regular path expressions* of edges to follow:

```
WHERE <db><lab>
          <name>$n</>
          <_*.city>$c</>
      </> ELEMENT_AS $l </> IN "myfile.xml"
```

is equivalent to path expressions:

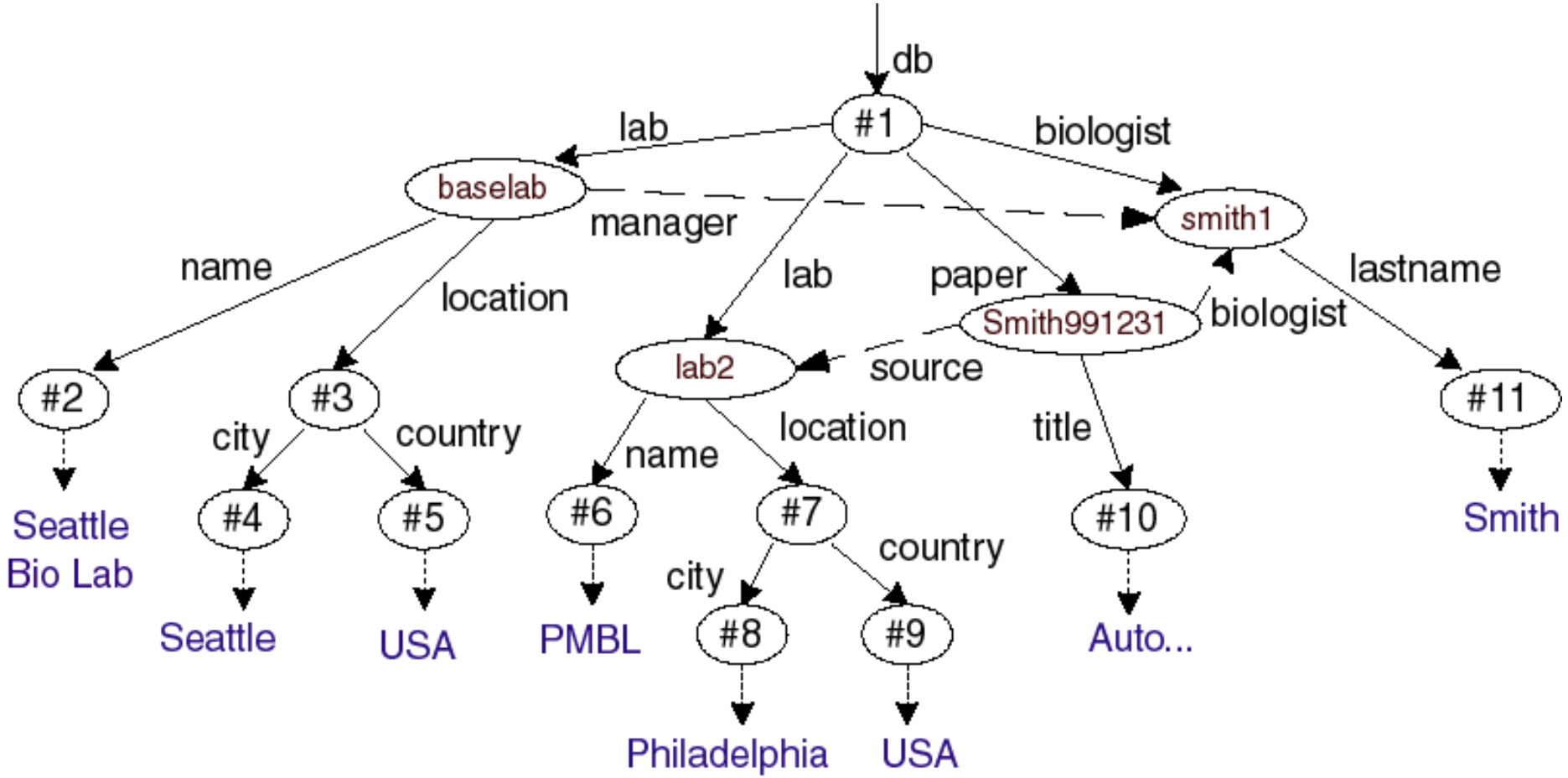
```
$l = (root).db.lab
$n = $l.name
$c = $l.(anything)*.city
```

- We want to find tuples of (`$l`, `$n`, `$c`) values
- Later we'll do relational-like operations on these tuples (e.g. join, select)

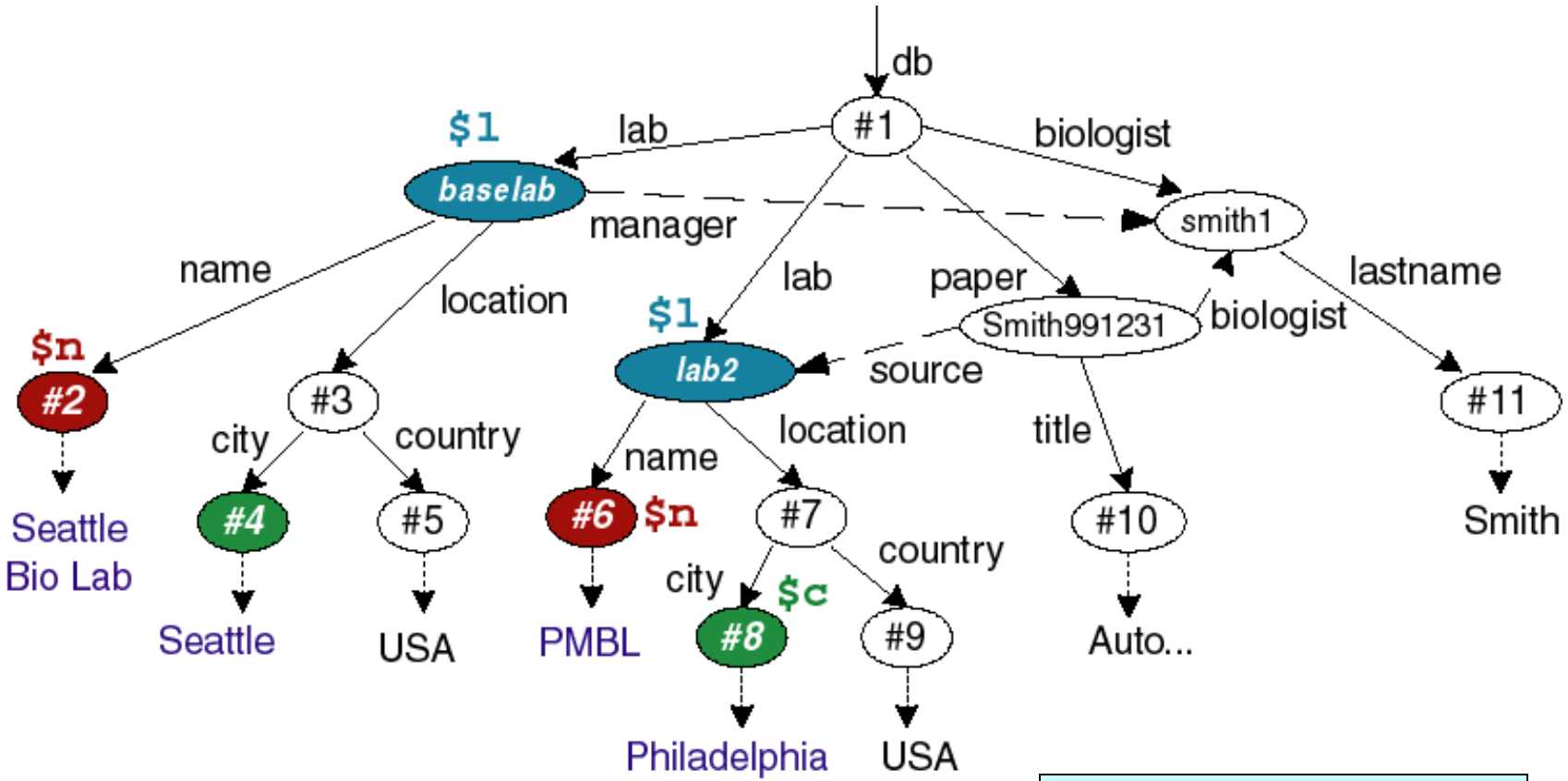
Example XML Document

```
<db>
  <lab ID="baselab" manager="smith1">
   <name>Seattle Bio Lab</name>
   <location>
     <city>Seattle</city>
     <country>USA</country>
   </location>
  </lab>
  <lab ID="lab2">
   <name>PMBL</name>
   <city>Philadelphia</city>
   <country>USA</country>
  </lab>
  <paper ID="Smith991231" source="baselab" biologist="smith1">
   <title>Autocatalysis of Spectral...</title>
  </paper>
  <biologist ID="smith1">
    <lastname>Smith</lastname>
  </biologist>
 </db>
```

XML Data Graph



Binding Graph Nodes to Variables




```
$l = (root).db.lab.name
$n = $l.name
$c = $l.(anything)*.city
```

l	n	c__
baselab	#2	#4
lab2	#6	#8

XML Operators

- New operators:
  - XML construction:
    - Create element (add tags around data)
    - Add attribute(s) to element (similar to join)
    - Nest element under other element (similar to join)
  - Path expression evaluation
    - X-scan

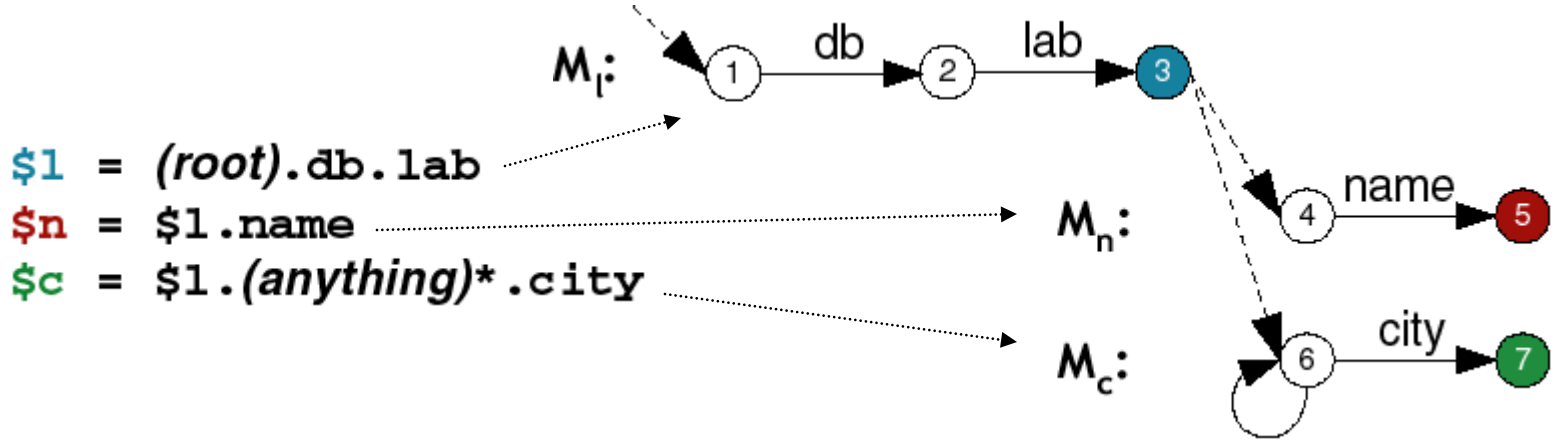
X-Scan: “Scan” for Streaming XML

- We often re-read XML from net on every query

  Data integration, data exchange, reading from Web
- Previous systems:
  - Store XML on disk, then index & query
  - Cannot amortize storage costs
- X-scan works on *streaming* XML data
  - Read & parse
  - Track nodes by ID
  - Index XML graph structure
  - **Evaluate path expressions to select nodes**

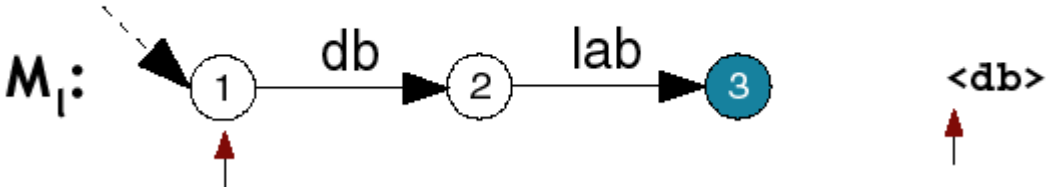
Computing Regular Path Expressions

Create *finite state machines* for path expressions



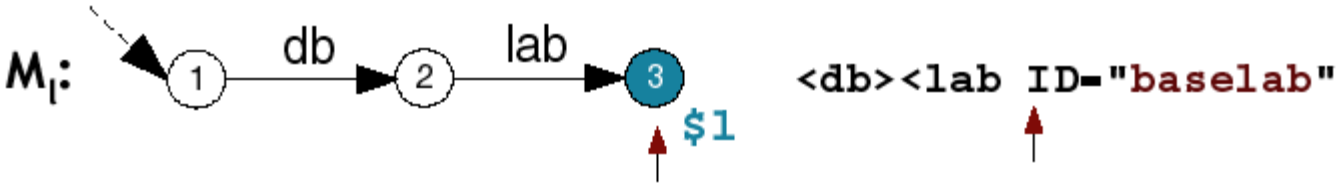
M_n and M_c are "dependent" on M_l

Start with initial (l) machine active:

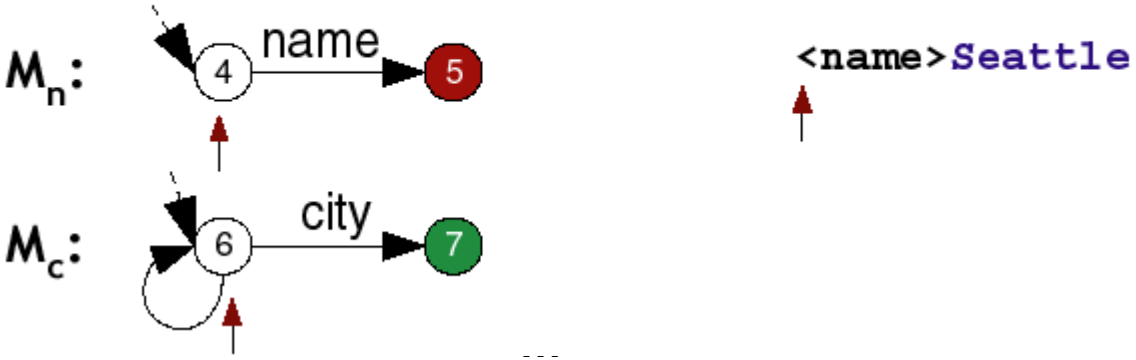


More State Machines

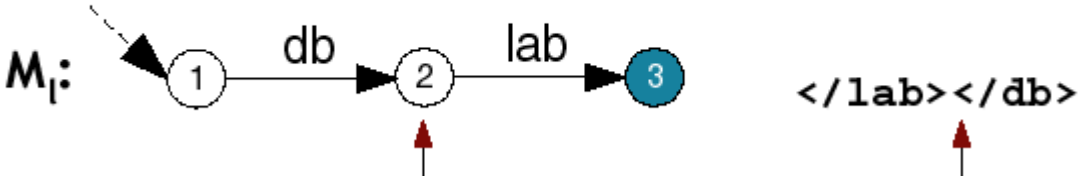
Read through XML to **`lab`** contents:



We now have a value for l; activate dependents:



Reset to previous state on each close-tag:



X-Scan works on *Graphs*

- The state machines work on trees – what about IDREFs?
  - Need to save the document so we can revisit nodes
  - Keep track of every ID
  - Build an “index” of the XML document’s structure; add *real* edges for every subelement and IDREF
  - When IDREF encountered, see if ID is known
    - If so, dereference and follow it
    - Otherwise, parse and index until we get to it, then process the newly indexed data

Recent Work in Execution

- XML query processors for data integration
  - Tukwila, Niagara (Wisconsin), Lore, MIX
- "Adaptive" query processing – smarter execution
  - Handling of exceptions (Tukwila)
  - Rescheduling of operations while delays occur (XJoin, query scrambling, Bouganim's multi-fragment execution)
  - Prioritization of tuples (WHIRL)
  - Rate-directed tuple flow (Eddies)
  - Partial results (Niagara)
  - "Continuous" queries (CQ, NiagraCQ)

Where's Execution Headed?

- *Adaptive scheduling* of operations – not purely iterator or data-driven
- *Robust* – as in distributed systems, exploit replicas, handle failures
- Able to show and update *partial/tentative results* – operators not "fully" blocking any more
- More *interactive and responsive* – many non-batch-oriented applications
- More *complex data models* – handle XML efficiently