

---

# DTS207TC Database Development and Design

## Lecture 4 XML

Chap 30 textbook

Di Zhang, Autumn 2025

*Page titles with \* will not be assessed*

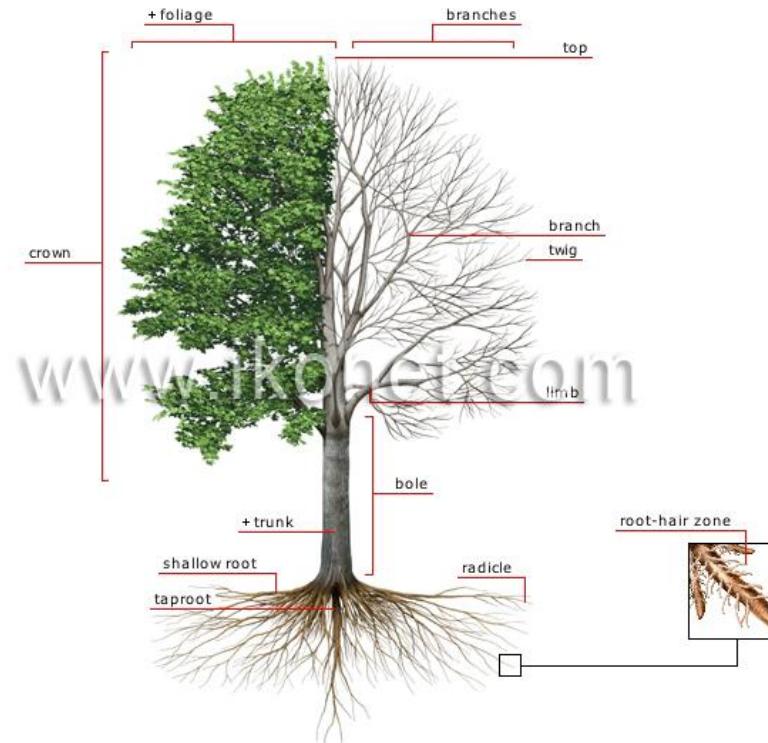
# \*Analog



- Possible patterns: Array, Tree, Graph. Nothing else!

# Outline

- Semi-Structured Data
- XML
  - Introduction
  - Structure
  - Schema
  - Query
  - API



# Semi-Structured Data

---

- Many applications require storage of complex data, whose schema changes often
- The relational model's requirement of atomic data types may be an overkill
  - E.g., storing set of interests as a set-valued attribute of a user profile may be simpler than normalizing it
- Data exchange can benefit greatly from semi-structured data
  - Exchange can be between applications, or between back-end and front-end of an application
  - Web-services are widely used today, with complex data fetched to the front-end and displayed using a mobile app or JavaScript

# Features of Semi-Structured Data Models

- **Flexible schema**

- **Wide column** representation: allow each tuple to have a different set of attributes, can add new attributes at any time
- **Sparse column** representation: schema has a fixed but large set of attributes, by each tuple may store only a subset

- **Multivalued data types**

- **Sets, multisets**
  - E.g.,: set of interests {'basketball', 'La Liga', 'cooking', 'anime', 'jazz'}

- **Key-value map** (or just **map** for short)
  - Store a set of key-value pairs
  - E.g., {(brand, Apple), (ID, MacBook Air), (size, 13), (color, silver)}
  - Operations on maps: *put(key, value)*, *get(key)*, *delete(key)*

- **Arrays**

- Widely used for scientific and monitoring applications

# Features of Semi-Structured Data Models

- **Arrays**
  - Widely used for scientific and monitoring applications
  - E.g., readings taken at regular intervals can be represented as array of values instead of (time, value) pairs
    - [5, 8, 9, 11] instead of {(1,5), (2, 8), (3, 9), (4, 11)}
- Multi-valued attribute types
  - Modeled using *non first-normal-form (NFNF)* data model
- Json

# 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.)

---

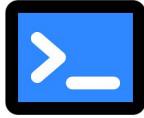
- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
  - Similar in concept to email headers
  - Does not allow for nested structures, no standard “type” language
  - Tied too closely to low level document structure (lines, spaces, etc)
- Each XML based 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

# 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



1

```
<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>
```

# 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

# Structure of XML Data (Cont.)

- 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

---

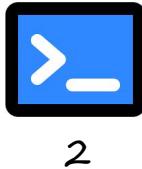
- Elements can have **attributes**

```
<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



- 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

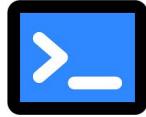
# Namespaces

---

- 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>
```

# More on XML Syntax

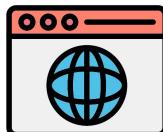


3

- 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 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



4

```
<!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

---

- 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 · · ·  
]>
```



5

# XML data with ID and IDREF attributes

```
<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>
```



6

# Limitations of DTDs

---

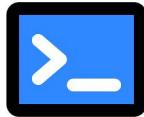
- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
  - Order is usually irrelevant in databases (unlike in the document-layout environment from which XML evolved)
  - $(A \mid B)^*$  allows specification 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

---

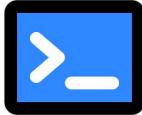
- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - Typing of values
    - E.g., integer, string, etc
    - Also, constraints on min/max values
  - User-defined, complex types
  - Many more features, including
    - uniqueness and foreign key constraints,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

# XML Schema Version of Univ. DTD



7

```
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xselement name="university" type="universityType" />
<xselement name="department">
    <xsccomplexType>
        <xsssequence>
            <xselement name="dept name" type="xs:string"/>
            <xselement name="building" type="xs:string"/>
            <xselement name="budget" type="xs:decimal"/>
        </xsssequence>
    </xsccomplexType>
</xselement>
...
<xselement name="instructor">
    <xsccomplexType>
        <xsssequence>
            <xselement name="ID" type="xs:string"/>
            <xselement name="name" type="xs:string"/>
            <xselement name="dept name" type="xs:string"/>
            <xselement name="salary" type="xs:decimal"/>
        </xsssequence>
    </xsccomplexType>
</xselement>
... Contd.
```



8

```
....  
<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>
```

# Querying 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 languages
  - XPath
    - Simple language consisting of path expressions
  - 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

# 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

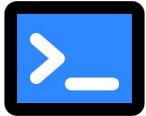
# XPath (Cont.)

---

- 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  $\geq 4$
  - IDREF attributes are not dereferenced automatically (more on this later)

# Functions in XPath

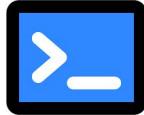
- XPath provides several functions
  - The function `count()` at the end of a path counts the number of elements in the set generated by the path
    - E.g., `/university-2/instructor[count(.//teaches/course)> 2]`
      - Returns instructors teaching more than 2 courses (on university-2 schema)
    - 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



10

# 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 form for specifying “all descendants”
  - “..” specifies the parent.



11

- 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 $courseld := $x/@course_id
where $x/credits > 3
return <course_id> { $courseld } </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



12

- 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

- 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

- 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>
```

# 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
    - Also provides functions for updating DOM tree



13

# Storage of XML Data

---

- XML data can be stored in
  - 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

---

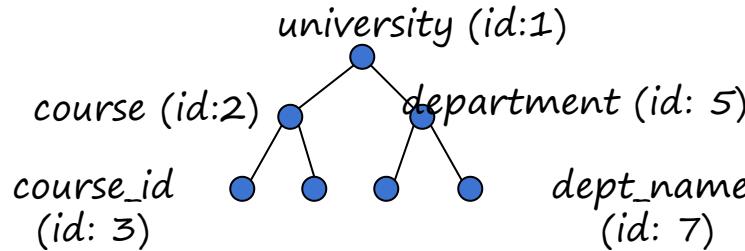
- Store each top level element as a string field of a tuple in a relational database
  - Use a single relation to store all elements, or
  - Use a separate relation for each top-level element type
    - E.g., account, customer, depositor relations
      - Each with a string-valued attribute to store 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 (Cont.)

- 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

- **Tree representation:** model XML data as tree and store using relations  
 $\text{nodes}(id, parent\_id, type, label, value)$



- 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

# Tree Representation (Cont.)

---

- 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 ( $i^{\text{th}}$  child) can be stored 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

- 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

- 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



14

```
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

---

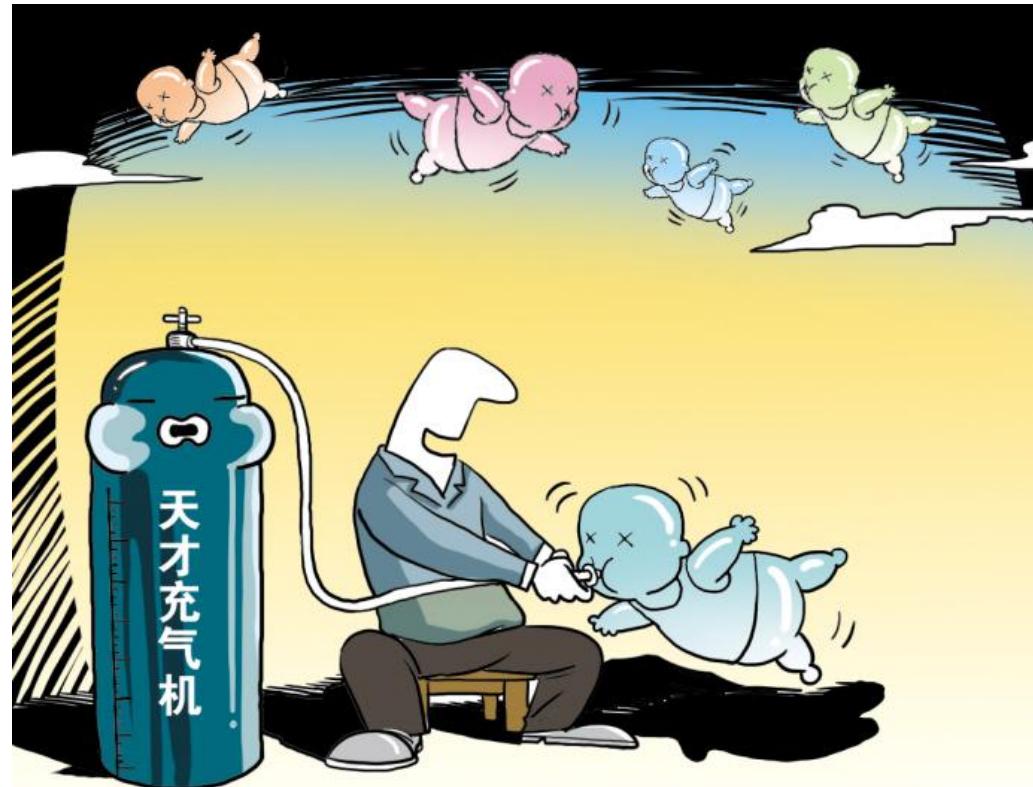
- Storing and exchanging data with complex structures
  - E.g., Open Document Format (ODF) format standard for storing Open Office and Office Open XML (OOXML) format standard for storing Microsoft Office documents
  - Numerous other standards for a variety of applications
    - ChemML, MathML
- Standard 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

# \*Pros and cons



Genius Inflator