Module 4: XML Representation

- Concepts
- Parsing and Validation
- Schemas
What is Metadata?

Literally, data about data

- Description of data that captures some useful property regarding its
  - Structure and meaning
  - Provenance: origins
  - Treatment as permitted or allowed: storage, representation, processing, presentation, or sharing

- Markup is metadata pertaining to media artifacts (documents, images), generally specified for suitable parsable units
Motivations for Metadata

Mediating information structure (surrogate for meaning) over time and space

- Storage: extend life of information
- Interoperation for business
- Interoperation (and storage) for regulatory reasons

General themes

- Make meaning of information explicit
- Enable reuse across applications: repurposing compare to screen-scraping
- Enable better tools to improve productivity

Reduce need for detailed prior agreements
Markup History

How much prior agreement do you need?

- No markup: significant prior agreement
- Comma Separated Values (CSV): no nesting
- Ad hoc tags
- SGML (Standard Generalized Markup Language): complex, few reliable tools; used for document management
- HTML (HyperText Markup Language): simplistic, fixed, unprincipled vocabulary that mixes structure and display
- XML (eXtensible Markup Language): simple, yet extensible subset of SGML to capture custom vocabularies
  - Machine processible
  - Comprehensible to people: easier debugging
Uses of XML

Supporting arms-length relationships

- Exchanging information across software components, even within an administrative domain
- Storing information in nonproprietary format
- Representing semistructured descriptions:
  - Products, services, catalogs
  - Contracts
  - Queries, requests, invocations, responses (as in SOAP): basis for Web services
<?xml version="1.0"?>  <!-- processing instruction -->
<topelem attr0="foo">  <!-- exactly one root -->
  <subelem attr1="v1" attr2="v2">
    Optional text (PCDATA)  <!-- parsed character data -->
    <subsubelem attr1="v1" attr2="v2"/>
  </subelem>
  <null_elem/>
</topelem>
<short_elem attr3="v3"/>
Exercise

Produce an example XML document corresponding to a directed graph
Compare with Lisp

List processing language

- S-expressions
- Cons pairs: car and cdr
- Lists as nil-terminated s-expressions
- Arbitrary structures built from few primitives
- Untyped
- Easy parsing
- Regularity of structure encourages recursion
Exercise

Produce an example XML document corresponding to

- An invoice from Locke Brothers for 100 units of door locks at $19.95, each ordered on 15 January and delivered to Custom Home Builders
- Factor in certified delivery via UPS for $200.00 on 18 January
- Factor in addresses and contact info for each party
- Factor in late payments
Meaning in XML

- Relational DBMSs work for highly structured information, but rely on column names for meaning
- Same problem in XML (reliance on names for meaning) but better connections to richer meaning representations
Because XML supports custom vocabularies and interoperation, there is a high risk of name collision.

A namespace is a collection of names.

Namespaces must be identical or disjoint:
- Crucial to support independent development of vocabularies
- MAC addresses
- Postal and telephone codes
- Vehicle identification numbers
- Domains as for the Internet
- On the Web, use URIs for uniqueness
XML Namespaces: 2

1. <!-- xml is reserved -->
   <?xml version="1.0"?>
   <arb1:top xmlns="a URI" <!-- default namespace -->
     xmlns:arb1="http://wherever.it.might.be/arb1-ns"
     xmlns:random="http://another.one/random-ns">

6. <arb1:aElem attr1="v1" attr2="v2">
   Optional text (PCDATA)
   <arb1:bElem attr1="v1" attr2="v2"/>
   </arb1:aElem>
   <random:simple_elem/>

11. <random:aElem attr3="v3"/>
    <!-- compare arb1:aElem -->
    </arb1:top>
Uniform Resource Identifier

- URIs are abstract
- What matters is their (purported) uniqueness
- URIs have no proper syntax per se

Kinds of URIs
- URLs, as in browsing: not used in standards any more
- URNs, which leave the mapping of names to locations up in the air

Good design: the URI resource exists
- Ideally, as a description of the resource in RDDL
- Use a URL or URN
RDDL

Resource Directory Description Language

- Meant to solve the problem that a URI may not have any real content, but people expect to see some (human readable) content
- Captures namespace description for people
  - XML Schema
  - Text description
Well-Formedness and Parsing

- An XML document maps to a parse tree (if well-formed; otherwise not XML)
  - Each element must end (exactly once): obvious nesting structure (one root)
  - An attribute can have at most one occurrence within an element; an attribute’s value must be a quoted string
- Well-formed XML documents can be parsed
XML InfoSet

A standardization of the low-level aspects of XML

- What an element looks like
- What an attribute looks like
- What comments and namespace references look like
- Ordering of attributes is irrelevant
- Representations of strings and characters

Primarily directed at tool vendors
Elements Versus Attributes: 1

- Elements are essential for XML: structure and expressiveness
  - Have subelements and attributes
  - Can be repeated
  - Loosely might correspond to independently existing entities
  - Can capture all there is to attributes
Elements Versus Attributes: 2

- Attributes are not essential
- End of the road: no subelements or attributes
- Like text; restricted to string values
- Guaranteed unique for each element
- Capture adjunct information about an element
- Great as references to elements

Good idea to use in such cases to improve readability
Elements Versus Attributes: 3

```xml
<invoice>
  <price currency='USD'>
    19.95
  </price>
</invoice>

Or

```xml
<invoice amount='19.95' currency='USD'/>
```

Or even

```xml
<invoice amount='USD 19.95'/>
```
Validating

Verifying whether a document matches a given grammar (assumes well-formedness)

- Applications have an explicit or implicit syntax (i.e., grammar) for their particular elements and attributes
  - Explicit is better have definitions
  - Best to refer to definitions in separate documents

- When docs are produced by external software components or by human intervention, they should be validated
Specifying Document Grammars

Verifying whether a document matches a given grammar

- Implicitly in the application
  - Worst possible solution, because it is difficult to develop and maintain

- Explicit in a formal document; languages include
  - Document Type Definition (DTD): in essence obsolete
  - XML Schema: good and prevalent
  - Relax NG: (supposedly) better but not as prevalent
XML Schema

- Same syntax as regular XML documents
- Local scoping of subelement names
- Incorporates namespaces
- (Data) Types
  - Primitive (built-in): string, integer, float, date, ID (key), IDREF (foreign key), ...
  - simpleType constructors: list, union
  - Restrictions: intervals, lengths, enumerations, regex patterns,
  - Flexible ordering of elements
- Key and referential integrity constraints
XML Schema: complexType

- Specifies types of elements with structure:
  - Must use a compositor if $\geq 1$ subelements
  - Subelements with types
  - Min and max occurrences (default 1) of subelements
- Elements with text content are easy
- EMPTY elements: easy
  - Example?
  - Compare to nulls, later
XML Schema: Compositors

- **Sequence**: ordered list
  - Can occur within other compositors
  - Allows varying min and max occurrence

- **All**: unordered
  - Must occur directly below root element
  - Max occurrence of each element is 1

- **Choice**: exclusive or
  - Can occur within other compositors
XML Schema: Main Namespaces

Part of the standard

- xsd: http://www.w3.org/2001/XMLSchema
  - Terms for defining schemas: schema, element, attribute, ...
  - The schema element has an attribute targetNamespace

- xsi: http://www.w3.org/2001/XMLSchema-instance
  - Terms for use in instances: schemaLocation, noNamespaceSchemaLocation, nil, type

- targetNamespace: user-defined
<Music xmlns="http://a.b.c/Muse"
    xmlns:xsi="the standard−xsi"
    xsi:schemaLocation="schema−URI schema−location−URL">
    <!-- Notice space character in above string -->
    ...
</Music>

Define null values as

<aElem xsi:nil="true"/>
XML Schema: Nillable

- An xsd:element declaration may state `nillable='true'`
  - An instance of the element might state `xsi:nil="true"`
  - The instance would be valid even if no content is present, even if content is required by default
Creating XML Schema Docs: 1

Included into the same namespace as the including doc

```xml
<xsd:schema xmlns:xsd="the-standard-xsd"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            targetNamespace="the-target">
    <include schemaLocation="part-one.xsd"/>
    <include schemaLocation="part-two.xsd"/>
</xsd:schema>
```

---

© Munindar P. Singh, CSC 513, Spring 2008
Creating XML Schema Docs: 2

- Use import instead of include
  - Imports may have different targets
  - Included schemas have the same target
  - Specify namespaces from which schemas are to be imported
  - Location of schemas not required and may be ignored if provided
Foreign Attributes in XML Schema

XML Schema elements allow attributes that are *foreign*, i.e., with a namespace other than the xsd namespace

- Must have an explicit namespace
- Can be used to insert any additional information, not interpreted by a processor
- Specific usage is with attributes from the xlink: namespace

```xml
<xsd:schema>
  <xsd:element name='course' type='cT'
    xlink:role='work' ncsu:offering='true'>
  4 </xsd:schema>
```

© Munindar P. Singh, CSC 513, Spring 2008
XML Schema Style Guidelines: 1

- Flatten the structure of the schema
- Don’t nest declarations as you would a desired instance document
- Make sure that element names are not reused
- Unqualified attributes cannot be global
- If dealing with legacy documents with the same element names having different meanings, place them in different namespaces where possible
- Use named types where appropriate
XML Schema Style Guidelines: 2

- Don’t have elements with mixed content
- Don’t have attribute values that need parsing
- Add unique IDs for information that may repeat
- Group information that may repeat
- Emphasize commonalities and reuse
  - Derive types from related types
  - Create attribute groups
xsd:annotation

- Should be the first subelement, except for the whole schema
- Container for two mixed-content subelements
  - xsd:documentation: for humans
  - xsd:appinfo: for machine-processible data
    - Such as application-specific metadata
    - Possibly using the Dublin Core vocabulary, which describes library content and other media
Module 5: XML Manipulation

Key XML query and manipulation languages include

- XPath
- XQuery
- XSLT
Metaphors for Handling XML: 1

How we conceptualize what XML documents are determines our approach for handling such documents

- **Text**: an XML document is text
  - Ignore any structure and perform simple pattern matches

- **Tags**: an XML document is text interspersed with tags
  - Treat each tag as an “event” during reading a document, as in SAX (Simple API for XML)
  - Construct regular expressions as in screen scraping
Metaphors for Handling XML: 2

- **Tree**: an XML document is a tree
  - Walk the tree using DOM (Document Object Model)

- **Template**: an XML document has regular structure
  - Let XPath, XSLT, XQuery do the work

- **Thought**: an XML document represents a graph structure
  - Access knowledge via RDF or OWL
XPath

Used as part of XPointer, SQL/XML, XQuery, and XSLT

- Models XML documents as trees with nodes
  - Elements
  - Attributes
  - Text (PCDATA)
  - Comments
  - Root node: above root of document
Achtung!

- Parent in XPath is like parent as traditionally in computer science

- Child in XPath is confusing:
  - An attribute is not a child of its parent
  - Makes a difference for recursion (e.g., in XSLT apply-templates)

- Our terminology follows computer science:
  - e-children, a-children, t-children
  - Sets via et-, ta-, and so on
相对或绝对的

类似于文件系统路径，但更为微妙

- 名称：需要向下走的元素
- 领导：/：根
- /：表示向下走一棵树
- .：当前匹配的（上下文）节点
- ../：父节点
XPath Location Paths: 2

- @attr: to check existence or access value of the given attribute
- text(): extract the text
- comment(): extract the comment
- [: generalized array accessors
- Variety of axes, discussed below
XPath Navigation

- Select children according to position, e.g., \([j]\), where \(j\) could be 1 \ldots last()
- Descendant-or-self operator, \(//\)
  - \(//\)elem finds all elems under the current node
  - \(//\)elem finds all elems in the document
- Wildcard, \(*\):
  - collects e-children (subelements) of the node where it is applied, but omits the t-children
  - \(@*\): finds all attribute values
XPath Queries (Selection Conditions)

- Attributes: //Song[@genre="jazz"]
- Text: //Song[starts-with(.//group, "Led")]
- Existence of attribute: //Song[@genre]
- Existence of subelement: //Song[group]
- Boolean operators: and, not, or
- Set operator: union (|), analogous to choice
- Arithmetic operators: >, <, . . .
- String functions: contains(), concat(), length(), starts-with(), ends-with()
- distinct-values()
- Aggregates: sum(), count()
XPath Axes: 1

Axes are addressable node sets based on the document tree and the current node

- Axes facilitate navigation of a tree
- Several are defined
- Mostly straightforward but some of them order the nodes as the reverse of others
- Some captured via special notation
  - current, child, parent, attribute, . . .
XPath Axes: 2

- preceding: nodes that precede the start of the context node (not ancestors, attributes, namespace nodes)
- following: nodes that follow the end of the context node (not descendants, attributes, namespace nodes)
- preceding-sibling: preceding nodes that are children of the same parent, in reverse document order
- following-sibling: following nodes that are children of the same parent
XPath Axes: 3

- ancestor: proper ancestors, i.e., element nodes (other than the context node) that contain the context node, in reverse document order
- descendant: proper descendants
- ancestor-or-self: ancestors, including self (if it matches the next condition)
- descendant-or-self: descendants, including self (if it matches the next condition)
XPath Axes: 4

- Longer syntax: child::Song
- Some captured via special notation
  - self::*:
  - child::node(): node() matches all nodes
  - preceding::*
  - descendant::text()
  - ancestor::Song
  - descendant-or-self::node(), which abbreviates to //
- Compare /descendant-or-self::Song[1] (first descendant Song) and //Song[1] (first Songs (children of their parents))
XPath Axes: 5

- Each axis has a principal node kind
  - attribute: attribute
  - namespace: namespace
  - All other axes: element
- * matches whatever is the principal node kind of the current axis
- node() matches all nodes
XPointer

Enables pointing to specific parts of documents

- Combines XPath with URLs
- URL to get to a document; XPath to walk down the document
- Can be used to formulate queries, e.g.,
  - Song-URL#xpointer(//Song[@genre="jazz"])
  - The part after # is a fragment identifier
- Fine-grained addressability enhances the Web architecture

High-level “conceptual” identification of node sets
XQuery

- The official query language for XML, now a W3C recommendation, as version 1.0
- Given a non-XML syntax, easier on the human eye than XML
- An XML rendition, XqueryX, is in the works
XQuery Basic Paradigm

The basic paradigm mimics the SQL (SELECT–FROM–WHERE) clause

```xquery
for $x in doc('q2.xml')//Song
where $x/@lg = 'en'
return
  <English-Sgr name='{$x/Sgr/@name}' ti='{$x/@ti}' />
```
FLWOR Expressions

Pronounced “flower”

- For: iterative binding of variables over range of values
- Let: one shot binding of variables over vector of values
- Where (optional)
- Order by (sort: optional)
- Return (required)

Need at least one of for or let
XQuery For Clause

The for clause

- Introduces one or more variables
- Generates possible bindings for each variable
- Acts as a mapping functor or iterator
  - In essence, all possible combinations of bindings are generated: like a Cartesian product in relational algebra
  - The bindings form an ordered list
XQuery Where Clause

The where clause

- Selects the combinations of bindings that are desired
- Behaves like the where clause in SQL, in essence producing a join based on the Cartesian product
XQuery Return Clause

The return clause

- Specifies what node-sets are returned based on the selected combinations of bindings
XQuery Let Clause

The let clause

- Like for, introduces one or more variables
- Like for, generates possible bindings for each variable
- Unlike for, generates the bindings as a list in one shot (no iteration)
XQuery Order By Clause

The order by clause

- Specifies how the vector of variable bindings is to be sorted before the return clause
- Sorting expressions can be nested by separating them with commas
- Variants allow specifying
  - descending or ascending (default)
  - empty greatest or empty least to accommodate empty elements
  - stable sorts: stable order by
  - collations: order by $t collation collation-URI: (obscure, so skip)
XQuery Positional Variables

The for clause can be enhanced with a positional variable

- A positional variable captures the position of the main variable in the given for clause with respect to the expression from which the main variable is generated

- Introduce a positional variable via the at $var construct
XQuery Declarations

The declare clause specifies things like

- **Namespaces**: declare namespace pref=’value’
  - Predefined prefixes include XML, XML Schema, XML Schema-Instance, XPath, and local

- **Settings**: declare boundary-space preserve (or strip)

- **Default collation**: a URI to be used for collation when no collation is specified
XQuery Quantification: 1

- Two quantifiers some and every
- Each quantifier expression evaluates to true or false
- Each quantifier introduces a bound variable, analogous to for

```xml
for $x$ in ...
where some $y$ in ...
satisfies $y$ ... $x$
return ...
```

Here the second $x$ refers to the *same* variable as the first
A typical useful quantified expression would use variables that were introduced outside of its scope

- The order of evaluation is implementation-dependent: enables optimization
- If some bindings produce errors, this can matter
- some: trivially false if no variable bindings are found that satisfy it
- every: trivially true if no variable bindings are found
Variables: Scoping, Bound, and Free

for, let, some, and every introduce variables

- The visibility variable follows typical scoping rules
- A variable referenced within a scope is
  - *Bound* if it is declared within the scope
  - *Free* if it not declared within the scope

```plaintext
for $x$ in ...
where some $x$ in ...
satisfies ...
return ...
```

Here the two $x$ refer to *different* variables
XQuery Conditionals

Like a classical if-then-else clause

- The else is not optional
- Empty sequences or node sets, written ( ), indicate that nothing is returned
XQuery Constructors

Braces { } to delimit expressions that are evaluated to generate the content to be included; analogous to macros

- **document { }**: to create a document node with the specified contents
- **element { } { }**: to create an element
  - **element foo { ’bar’ }**: creates `<foo>Bar</foo>`
  - **element { ’foo’ } { ’bar’ }**: also evaluates the name expression
- **attribute { } { }**: likewise
- **text { body}**: simpler, because anonymous
XQuery Effective Boolean Value

Analogous to Lisp, a general value can be treated as if it were a Boolean

- A xs:boolean value maps to itself
- Empty sequence maps to false
- Sequence whose first member is a node maps to true
- A numeric that is 0, negative, or NaN maps to false, else true
- An empty string maps to false, others to true
Defining Functions

```xml
declare function local:itemftop($t)
{
    local:itemf($t,())
};
```

- Here `local:` is the namespace of the query
- The arguments are specified in parentheses
- All of XQuery may be used within the defining braces
- Such functions can be used in place of XPath expressions
Functions with Types

declare function local:itemftop($t as element()) as element()
{
    local:itemf($t,())
};

■ Return types as above
■ Also possible for parameters, but ignore such for this course
XSLT

A programming language with a functional flavor

- Specifies (stylesheet) transforms from documents to documents
- Can be included in a document (best not to)

```xml
<?xml version="1.0"?>
<?xml−stylesheet type="text/xsl" href="URL−to−xsl−sheet"?>
<main−element>
  5  ...
</main−element>
```
XQuery versus XSLT: 1

Competitors in some ways, but

- Share a basis in XPath
- Consequently share the same data model
- Same type systems (in the type-sensitive versions)
- XSLT got out first and has a sizable following, but XQuery has strong backing among vendors and researchers
XQuery versus XSLT: 2

- XQuery is geared for querying databases
  - Supported by major relational DBMS vendors in their XML offerings
  - Supported by native XML DBMSs
  - Offers superior coverage of processing joins
  - Is more logical (like SQL) and potentially more optimizable

- XSLT is geared for transforming documents
  - Is functional rather than declarative
  - Based on template matching
There is a bit of an arms race between them

- **Types**
  - XSLT 1.0 didn’t support types
  - XQuery 1.0 does
  - XSLT 2.0 does too

- XQuery presumably will be enhanced with capabilities to make updates, but XSLT could too
XSLT Stylesheets

A programming language that follows XML syntax

- Use the XSLT namespace (conventionally abbreviated xsl)
- Includes a large number of primitives, especially:
  - <copy-of> (deep copy)
  - <copy> (shallow copy)
  - <value-of>
  - <for-each select="...">
  - <if test="...">
  - <choose>
XSLT Templates: 1

A pattern to specify where the given transform should apply: an XPath expression

This match only works on the root:

```xml
<xsl:template match="/">
  ...
</xsl:template>
```

Example: Duplicate text in an element

```xml
<xsl:template match="text()">
  2  
  <xsl:value-of select='.' />
  <xsl:value-of select='.' />
</xsl:template>
```
XSLT Templates: 2

- If no pattern is specified, apply recursively on et-children via `<xsl:apply-templates/>`
- By default, if no other template matches, recursively apply to et-children of current node (ignores attributes) and to root:

```xml
<xsl:template match="* | /">
  <xsl:apply-templates/>
</xsl:template>
```
XSLT Templates: 3

- Copy text node by default
- Use an empty template to override the default:

```xml
<xsl:template match="X"/>
```

2 <!-- X = desired pattern -->

Confine ourselves to the examples discussed in class (ignore explicit priorities, for example)
XSLT Templates: 4

- Templates can be named
- Templates can have parameters
  - Values for parameters are supplied at invocation
  - Empty node sets by default
  - Additional parameters are ignored
XSLT Variables

- Explicitly declared
- Values are node sets
- Convenient way to document templates
Document Object Model (DOM)

Basis for parsing XML, which provides a node-labeled tree in its API

- Conceptually simple: traverse by requesting element, its attribute values, and its children
- Processing program reflects document structure, as in recursive descent
- Can edit documents
- Inefficient for large documents: parses them first entirely even if a tiny part is needed
- Can validate with respect to a schema
DOM Example

```java
DOMParser p = new DOMParser();
p.parse("filename");

Document d = p.getDocument();
Element s = d.getDocumentElement();
NodeList l = s.getElementsByTagName("member");
Element m = (Element) l.item(0);
int code = m.getAttribute("code");

NodeList kids = m.getChildNodes();
Node kid = kids.item(0);
String elemName = ((Element) kid).getTagName(); ...
Simple API for XML (SAX)

- Parser generates a sequence of events: startElement, endElement, . . .
- Programmer implements these as *callbacks*
  - More control for the programmer
- Processing program does not necessarily reflect document structure
class MemberProcess extends DefaultHandler {
    public void startElement(String uri, String n,
        String qName, Attributes attrs) {
        if (n.equals("member")) code = attrs.getValue("code");
        if (n.equals("project")) inProject = true;
        buffer.reset();
    }
}

...
SAX Example: 2

1

...  

public void endElement (String uri, String n,  
String qName) {

6  if (n.equals("project")) inProject = false;
   if (n.equals("member") && !inProject)
      ... do something ...  
}
}
SAX Filters

A component that mediates between an XMLReader (parser) and a client

- A filter would present a modified set of events to the client

- Typical uses:
  - Make minor modifications to the structure
  - Search for patterns efficiently
    - What kinds of patterns, though?
  - Ideally modularize treatment of different event patterns
  - In general, a filter can alter the structure of the document
Creating XML from Legacy Sources

Often need to read in information from non-XML sources

- From relational databases
  - Easier because of structure
  - Supported by vendor tools

- From flat files, CSV documents, HTML Web pages
  - Bit of a black art: lots of heuristics
  - Tools based on regular expressions
Programming with XML

- Limitations
  - Difficult to construct and maintain documents
  - Internal structures are cumbersome; hence the criticisms of DOM parsers

- Emerging approaches provide superior binding from XML to
  - Programming languages
  - Relational databases

- Check pull-based versus push-based parsers
Module 6: XML Storage

The major aspects of storing XML include

- XML Keys
- Concepts: Data and Document Centrism
- Storage
- Mapping to relational schemas
- SQL/XML
Integrity Constraints in XML

- Entity: xsd:unique and xsd:key
- Referential: xsd:keyref
- Data type: XML Schema specifications
- Value: Solve custom queries using XPath or XQuery

Entity and referential constraints are based on XPath
XML Keys: 1

Keys serve as generalized identifiers, and are captured via XML Schema elements:

- **Unique**: candidate key
  - The selected elements yield unique field tuples

- **Key**: primary key, which means candidate key plus
  - The tuples exist for each selected element

- **Keyref**: foreign key
  - Each tuple of fields of a selected element corresponds to an element in the referenced key
Two subelements built using restricted application of XPath from within XML Schema

- **Selector**: specify a set of objects: this is the scope over which uniqueness applies

- **Field**: specify what is unique for each member of the above set: this is the identifier within the targeted scope
  - Multiple fields are treated as ordered to produce a tuple of values for each member of the set
  - The order matters for matching keyref to key
Selector XPath Expression

A selector finds descendant elements of the context node

- The sublanguage of XPath used *allows*
  - Children via ./child or ./* or child
  - Descendants via .// (not within a path)
  - Choice via |
- The subset of XPath used *does not allow*
  - Parents or ancestors
  - text()
  - Attributes
  - Fancy axes such as preceding, preceding-sibling, . . .
Field XPath Expression

A field finds a unique descendant element (simple type only) or attribute of the context node

- The subset of XPath used allows
  - Children via ./child or ./*
  - Descendants via .// (not within a path)
  - Choice via |
  - Attributes via @attribute or @*

- The subset of XPath used does not allow
  - Parents or ancestors
  - text()
  - Fancy axes such as preceding, . . .

An element yields its text()
XML Foreign Keys

<keyref name="..." refer="primary-key-name">
  <selector xpath="..."/>
  <field name="..."/>
</keyref>

- Relational requirement: foreign keys don’t have to be unique or non-null, but if one component is null, then all components must be null.
Placing Keys in Schemas

- Keys are associated with elements, not with types.
- Thus the . in a key selector expression is bound.
- Could have been (but are not) associated with types where the . could be bound to whichever element was an instance of the type.
Data-Centric View: 1

1 <relation name='Student'>
  <tuple>
    <attr1>V11</attr1>
    ...
    <attrn>V1n</attrn>
  </tuple>
</relation>

- Extract and store via mapping to DB model
- Regular, homogeneous structure
Data-Centric View: 2

- Ideally, no mixed content: an element contains text or subelements, not both
- Any mixed content would be templatic, i.e.,
  - Generated from a database via suitable transformations
  - Generated via a form that a user or an application fills out
- Order among siblings likely irrelevant (as is order among relational columns)

Expensive if documents are repeatedly parsed and instantiated
Document-Centric View

- Irregular: doesn’t map well to a relation
- Heterogeneous data
- Depending on entire doc for application-specific meaning
Data- vs Document-Centric Views

- **Data-centric**: data is the main thing
  - XML simply renders the data for transport
  - Store as data
  - Convert to/from XML as needed
  - The structure is important

- **Document-centric**: documents are the main thing
  - Documents are complex (e.g., design documents) and irregular
  - Store documents wherever
  - Use DBMS where it facilitates performing important searches
Storing Documents in Databases

- Use character large objects (CLOBs) within DB: searchable only as text
- Store paths to external files containing docs
  - Simple, but no support for integrity
- Use some structured elements for easy search as well as unstructured clobes or files
- Heterogeneity complicates mappings to typed OO programming languages

Storing documents in their entirety may sometimes be necessary for external reasons, such as regulatory compliance
Database Features

- Storage: schema definition language
- Querying: query language
- Transactions: concurrency
- Recovery
Potential DBMS Types for XML: 1

- **Object-oriented**
  - Nice structure
  - Intellectual basis of many XML concepts, including schema representations and path expressions
  - Not highly popular in standalone products

- **Relational**
  - Limited structuring ability (1NF: each cell is atomic)
  - Extremely popular
  - Well optimized for flat queries
Potential DBMS Types for XML: 2

- Object relational: hybrids of above
  - Not highly popular in standalone products
- Custom XML stores or native XML databases
  - Emerging ideas: may lack core database features (e.g., recovery, ...)  
  - Enable fancier content management systems
- Leading open source products:
  - Apache Xindice (server; XPath)
  - Berkeley DB XML (libraries; XQuery)
XML to Relational Databases

- Using large objects
- Flatten XML structures
- Referring to external files

Recall that for a relational schema, its entire set of attributes is necessarily a superkey
Artificial Representation: Repetitious

Capturing an object hierarchy in a relation

- Imagine an artificial identifier for each node
- Construct a relation with three main relational attributes or columns
  - One column for the identifier
  - One column for the name of an attribute (i.e., element name)
  - One column for the value (assumes the value would fit into the same relational type: potentially this could be CLOB or BLOB)
Artificial Representation: Graph

Use four generic relations to represent a graph

- Vertices:
  - Element ID, Name
- Contents
  - Element ID, Text, number (to allow multiple text nodes)
- Attributes
  - ID, Attribute name, Attribute value
- Edges
  - Source ID, Target ID

Better typed than repetitious style because this has no nulls
The “natural” approaches are based on tuple-generating elements (TGEs)

- Choose one XML element type as the TGE
  - TGE corresponds to a tuple
  - The key is based on an ID attribute or text of the TGE
- A relational attribute (column) for each subelement or attribute
- Easiest if there is an attribute for IDs and there are no other attributes
Shallow Representation: 2

- **Consequences**
  - Nulls for missing subelements can proliferate
  - Subelements with structure (subelements or attributes) aren’t represented well
  - Ancestors cannot be searched for
Deep Representation

Also called *shredding* an XML document

- Choose a TGE as before
- A column for each descendant, except that
  - Can skip *wrapper* elements (no text, only subelements), but must reconstruct them to create an XML document

Consequences

- Nulls for missing subelements
- Lots of columns in a relation
- Ancestors cannot be searched for
- Loses structural information
Representing Ancestors

Ancestors are the elements that are above the scope of the given TGE

- Choose a TGE as before
- A column for each descendant as before
- A column for each ancestor (that needs to be searched)
  - Appropriate attributes or text fields to make the search worthwhile

- Consequences
  - Nulls for missing subelements
  - Lots of columns in a relation
Generalized TGE

- Each element is a TGE, yielding a different relation
- A column for each terminal child: attribute or text
- A column for each ancestor to capture the entire path from root to this node
  - Must promote uniquifying content so that each TGE yields unique tuples

Consequences
- Nulls for missing subelements
- Lots of relations
- Lots of columns in a relation
Variations in Structure

- Create separate relations for each variant

Consequences

- Lots of possible structures to store
- Queries would not be succinct
- Acceptable only if we know in advance that the number of variants is small and the data in each is substantial
Semistructured Representation

Create two (sets of) relations

- **Specific part:** one (or more) relations based on one of the natural approaches
- **Generic part:** one relation based on an artificial approach
Thoughtful Design

- The above approaches are not sensitive to the meaning and motivation behind the XML structure
- Understand the XML structure via a conceptual model (in terms of entities and relationships)
- Avoid unnecessary nesting in the XML structure, if possible
- Design a corresponding relational schema by hand

This is not always possible, though
Evaluation

How does the above work for data-centric and document-centric views?

- Compare with respect to
  - Document structure
  - Document “roundtripping” (compare &, &amp;, #a39)
  - Normalization

- Are the documents unique?
- Are the documents unique up to “isomorphism”?
Schema Evolution

A big problem for databases in practical settings

- For relational schemas, certain kinds of updates are simpler than others
- Can have consequences on optimization
- XML schemas can be evolved by using XSLT to map old data to new schema
From Relations to XML

Mapping a relation schema (set of relations plus functional dependencies) to an XML document

- Map relation $R$ to an element $R_E$ with key or unique constraints
- Map column $C$ of $R$ to an attribute of $R_E$ or equivalently a child element with just text
- Map relation $S$ with a foreign key to $R$ to
  - A child element $S_E$ of $R_E$ (omit foreign key content from $S_E$): works if only one such $R_E$ for $S_E$; OR
  - An element $S_E$ that includes the foreign key content, and includes a keyref to $R_E$
Null Value: 1

A special value, not in any domain, but combinable with any domain

- Need?
- Possible meanings
  - Not applicable
  - Unknown: missing
  - Questionable existence
  - Absent (known but absent)
- Hazards of null values?
Null Value: 2

XML Schema enables developing custom null values for each domain

- Create an arbitrary value that
  - Matches the given data type
  - Is not a valid value of the domain, however

- Design applications to understand specific restricted type
XML Schema Null

- `<elem/>` (equivalently `<elem></elem>`) means that the element contains the empty string
  - This is not null
- `xsi` defines the attribute `nil`
  - Used as `<elem xsi:nil="true"/>` if elem is declared nillable (via nillable="true")
Structured Query Language

- **Data Definition Language**: CREATE TABLE
- **Data Manipulation Language**: SELECT, INSERT, DELETE, UPDATE
- **Basic paradigm for SELECT**

```
SELECT t1.column_1, t1.column_2 ... tm.column_n
FROM table_1 t1, table_m tm
WHERE t1.column_3=t4.column_4 AND ...
```