Documents vs. Databases

- Documents are typically small, while databases can be large.
- Documents are usually static, whereas databases are typically dynamic.
- A document has an implicit structure, while a database has an explicit structure.
- Documents are usually semi-structured (without an explicit type), while databases are structured, constrained by a schema.
- Documents are human friendly, while databases are machine friendly.
- Concerns about documents include presentation, editing, character encoding, language; while databases focus on models, queries, concurrency control, performance.
Why study XML?

- Huge demands for data exchange
  - Across platforms
  - Across enterprises
- Huge demands for data integration
  - Heterogeneous data sources
  - Data sources distributed across different locations
- XML (eXtensible Markup Language) has become the prime standard for data exchange on the Web and a uniform data model for data integration.
What is wrong with HTML?

HTML (HyperText Markup Language)

A minor format change to the HTML document may break the parser – and yield wrong answer to the query

Why? HTML tags are

 ✓ predefined and fixed
 ✓ describing display format rather than structure of data

HTML is good for presentation (human friendly), but does not help automatic data extraction by means of programs
An XML solution

XML (eXtensible Markup Language):

```
<book>
    <title>Database Systems</title>
    <author>R. Ramakrishnan</author>
    <year>1999</year>
</book>

<book id="B2">
    <title>Complexity Theory</title>
    <author>C. Papadimitriou</author>
</book>
```

...
XML vs. HTML

✓ XML tags:
  • user-defined
  • describing the structure of the data
XML is both human friendly and computer friendly.

✓ HTML is human friendly but not computer friendly;

HTML tags:
  • predefined and fixed
  • describing display format rather than structure of data
    indented for human consumption
What we shall in this course

✓ XML basics: elements, attributes, tree model
✓ Document Type Definition
  – “types”: element type definition
  – “constraints”: ID/IDREF
✓ XML query Languages
  – XPath
  – XQuery, XSLT
✓ To learn XML properly, take QSX!
History: SGML, HTML, XML

SGML: Standard Generalized Markup Language
    -- Charles Goldfarb, ISO 8879, 1986
✓ DTD (Document Type Definition)
✓ powerful and flexible tool for structuring information, but
  – complete, generic implementation of SGML proven extremely difficult
  – tools for working with SGML documents proven expensive
✓ two sub-languages that have outpaced SGML:
From HTML to XML

HTML is good for presentation (human friendly), but does not help automatic data extraction by means of programs (not computer friendly).

Why? HTML tags:
✓ predefined and fixed
✓ describing display format, not the structure of the data.

<h3>John Smith</h3>
<b>Taking CS 101</b> <br>
<em>GPA: 1.5</em> <br>
<h3>CS 101</h3>
<b>Intro to CS</b>
XML tags:
✓ user defined
✓ describing the structure of the data

```xml
<school>
  <student id = "011">
    <name>
      <firstName>John</firstName>   <lastName>Smith</lastName>
    </name>
    <taking> CS 101 </taking>
    <GPA> 1.5 </GPA>
  </student>
  <course cno = "CS 101">
    <title> Intro to CS </title>
  </course>
</school>
```
XML vs. HTML

- user-defined new tags, describing structure instead of display
- structures can be arbitrarily nested (even recursively defined)
- optional description of its grammar (DTD) and thus validation is possible

What is XML for?
- The prime standard for data exchange on the Web
- A uniform data model for data integration

XML presentation:
- XML standard does not define how data should be displayed
XML consists of tags and text

```xml
<course cno = "CS 101">
  <title>Intro to CS</title>
</course>
```

- tags come in pairs: markups
  - start tag, e.g., `<course>`
  - end tag, e.g., `</course>`

- tags must be properly nested
  - `<course> <title> ... </title> </course>` -- good
  - `<course> <title> ... </title> </course>` -- bad

- XML has only one “basic” type: text, called PCDATA (Parsed Character DATA)
XML Elements

- Element: the segment between an start and its corresponding end tag
- subelement: the relation between an element and its component elements.

```xml
<person>
  <name>John Smith</name>
  <email>john.smith@abc.com</email>
  <oldemail>j.smith@abc.com</oldemail>
  <oldemail>john.smith@xyz.com</oldemail>
</person>
```
Special elements

- **root element**: an XML document consists of a single element called the **root** element, e.g.,
  
  <db>
      <person> ... </person>
      <person> ... </person> ...
  </db>

- **empty** element: special element indicating non-textual content,
  - `<foo></foo>` or simply `<foo/>`
  - an element may carry attributes
    
    `<image img="picture.gif" />`

  to be interpreted by applications
XML attributes

A start tag may contain attributes describing certain “properties” of the element (e.g., dimension or type)

```xml
<picture>
  <height dim="cm">2400</height>
  <width dim="in">96</width>
  <data encoding="gif">M05-C$ …</data>
</picture>
```

References (meaningful only when a DTD is present):

```xml
<person id="011" pal="012">
  <name>John Smith</name>
</person>
<person id="012" pal="011">
  <name>Mary Brown</name>
</person>
```
The "structure" of XML attributes

✓ XML attributes cannot be nested -- flat
✓ the names of XML attributes of an element must be unique.
   one can't write  <person pal="John" pal="Mary"> ...
✓ XML attributes are not ordered
   <person id="011" pal="012">
     <name> John Smith</name>
   </person>

is the same as

   <person pal="012" id="011">
     <name> John Smith</name>
   </person>

✓ Attributes vs. subelements: unordered vs. ordered, and
  – attributes cannot be nested (flat structure)
  – subelements cannot represent references
Well-formed XML documents

A document is well-formed if it satisfies two constraints (when only elements and attributes are considered):

✓ Tags have to nest properly
✓ Attributes have to be unique

Very weak constraints: it does little more than ensure that XML data will parse into a labeled tree
<?xml version='1.0'?>
<!DOCTYPE book PUBLIC "~/school.dtd">
<?xml:stylesheet href="school.xsl" type="text/xsl"?>
<school>
    <!-- school database -->
    <student id="011">
        <name>
            <firstName>John</firstName>
            <lastName>Smith</lastName>
        </name>
        <taking>CS 101</taking>
        <GPA>1.5</GPA>
    </student>
    <course cno="CS 101">
        <title>Intro to CS</title>
    </course>
</school>
The XML tree model

An XML document is modeled as a node-labeled ordered tree.

- **Element** node: typically internal, with a name (tag) and children (subelements and attributes), e.g., `student`, `name`.
- **Attribute** node: leaf with a name (tag) and text, e.g., `@id`.
- **Text** node: leaf with text (string) but without a name.

```
<db>
  <student>
    <id>123</id>
    <name>John</name>
    <lastName>Smith</lastName>
  </student>
  <student>
    <name>David</name>
    <lastName>Lee</lastName>
  </student>
  <course cno="CS 101">
    <title>Intro to CS</title>
  </course>
</db>
```
Query Languages for XML

- XPath
- XSLT
- XQuery
Common Querying Tasks

✓ Filter, select XML values
  – Navigation, selection, extraction
✓ Merge, integrate values from multiple XML sources
  – Joins, aggregation
✓ Transform XML values from one schema to another
  – XML construction
Query Languages

✓ XPath
  – Common language for navigation, selection, extraction
  – Used in XSLT, XQuery, XML Schema, . . .
✓ XSLT: XML ⇒ XML, HTML, Text
  – Loosely-typed scripting language
  – Format XML in HTML for display in browser
  – Highly tolerant of variability/errors in data
✓ XQuery 1.0: XML ⇒ XML
  – Strongly-typed query language
  – Large-scale database access
  – Safety/correctness of operations on data
XML data

<book year="1996">
  <title>HTML</title>
  <author><last>Lee</last> <first>T.</first></author>
  <author><last>Smith</last> <first>C.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price>59.99</price>
</book>

<book year="1999">
  <title>Database Systems</title>
  <author><last>Ramakrishnan</last>
            <first>Raghu</first></author>
  <publisher>white house</publisher>
</book>
<!ELEMENT bib (book*) >
<!ELEMENT book (title, (author+ | editor+),
    publisher?, price?) >
<!ATTLIST book year CDATA #required >
<!ELEMENT author (last, first)>  
<!ELEMENT editor (last, first, affiliation)> 
<!ELEMENT publisher (#PCDATA) >
....
Data model

Node-labeled, ordered tree

bib

book
title author author publisher phone @year

last first last first

@year title author publisher

last first
XPath

W3C standard: [www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)

- Navigating an XML tree and finding parts of the tree (node selection and value extraction)

  Given an XML tree $T$ and a context node $n$, an XPath query $Q$ returns
  
  - the set of nodes reachable via $Q$ from the node $n$ in $T$ – if $Q$ is a binary query
  - truth value indicating whether $Q$ is true at $n$ in $T$ – if $Q$ is a Boolean query.

- Implementations: XALAN, SAXON, Berkeley DB XML – freeware, which you can play with

- A major element of XSLT, XQuery and XML Schema

- XPath 2.0 (Turing-Complete)
XPath constructs

XPath query Q:
- Tree traversal: downward, upward, sideways
- Relational/Boolean expressions: qualifiers (predicates)
- Functions: aggregation (e.g., count), string functions

//author[last="Ramakrishnan"]
Downward traversal

Syntax:
Q ::= . | l | @l | Q/Q | Q | Q | //Q | /Q | Q[q]
q ::= Q | Q op c | q and q | q or q | not(q)

✓ .: self, the current node
✓ l: either a tag (label) or *: wildcard that matches any label
✓ @l: attribute
✓ /, |: concatenation (child), union
✓ //: descendants or self, “recursion”
✓ [q]: qualifier (filter, predicate)
  – op: =, !=, <=, <, >, >=, >
  – c: constant
  – and, or, not(): conjunction, disjunction, negation

Existential semantics: /bib/book[author/last="Ramakrishnan"]
Examples:

- parent/child: /bib/book
- ancestor//descendant: bib//last, //last
- wild card: bib/book/*
- attributes: bib/book/@year
- attributes with wild cards: //book/@*
- union: book/(editor | author)

Are book/(editor | author) and //(editor | author) “equivalent” at context nodes (1) root, (2) book, (3) author?
Filters (qualifiers)

✓ //book[price]/title -- titles of books with a price
✓ //book[title and author and not(price)]/title
titles of books with authors, title but no price
✓ //book[author/last = “Ramakrishnan”]/title
titles of books with an author whose last name is Ramakrishnan
✓ //book[editor | author]/title
titles of books with with either an author or an editor
Upward traversal

Syntax:

\[ Q ::= \ldots | ../Q | \text{ancestor} :: Q | \text{ancestor-or-self} :: Q \]

- \( ../ \): parent
- \text{ancestor, ancestor-or-self}: recursion

Example:

- \( //\text{author}[../\text{title} = \text{"Databases"}]/\text{last} \)
  find last names of authors of books with the title “Databases”

- \text{ancestor} :: \text{book}[:,//\text{last} = \text{"Ramakrishnan"}]\]
  find book ancestors with “Ramakrishnan” as its last descendant
Sideways

Syntax:
Q ::= . . . | following-sibling ::=Q | preceding-sibling ::=Q

✓ following-sibling: the next sibling
✓ preceding-sibling: the previous sibling
✓ position function: e.g., //author[position( ) < 2]

Example:
✓ following-sibling :: book [//last=“Ramakrishnan”]
  find the next book written by Ramakrishnan
✓ preceding-sibling :: book[//last=“Ramakrishnan”]
  find the last book written by Ramakrishnan
Query Languages for XML

✓ XPath
✓ XSLT
✓ XQuery
XSL (eXtensible Stylesheet Language)

W3C recommendation  [www.w3.org/Style/XSL](http://www.w3.org/Style/XSL)

- Two separate languages:
  - XSLT: transformation language, Turing complete
  - a formatting language

- Purpose: stylesheet specification language
  - displaying XML documents: XML -> HTML
  - transforming/querying XML data: XML -> XML

- Implementations: SAXON, XALAN, …
XSL programs

XSL program: a collection of template rules
✓ template rule = pattern + template
✓ computation:
  – starts from the root
  – apply a pattern to each node. If it matches, execute the corresponding template (to construct XML/HTML), and apply templates recursively on its children.
✓ patterns:
  – match pattern: determine content – whether or not to apply the rule?
  – select pattern: identify nodes to be processed, set of nodes
An example XSLT program

Q1: Find titles and authors of all books published by Addison-Wesley after 1991.

```xml
<xsl:template match="/bib/book[＠year ＞ 1991 and publisher='Addison-Wesley']">
  <result>
    <title> <xsl:value-of select="title"/> </title>
    <xsl:for-each select="author"/>
    <author><xsl:value-of/> </author>
  </xsl:for-each>
</result>
</xsl:template>
```
Basic XSLT constructs

✓ a collection of templates: `<xsl:template>`
✓ match pattern: `match="bib/book[@year > 1991 and publisher='Addison-Wesley']"`
✓ select pattern: `select="title", xsl:for-each select="author"`
✓ `value-of`: `string`
✓ constructing XML data:
  `<result>`
    `<title> <xsl:value-of select="title" /> </title>`
  ...
  `</result>`
Patterns

- match pattern: (downward) XPath
  - parent/child: bib/book
  - ancestor//descendant (_*): bib//last, //last, ...

- select patterns: XPath

Example:

```xml
<xsl:template match="/bib/book/title">
  <result>
    <title> <xsl:value-of /> </title>
    <author> <xsl:value-of select="../author" /> </author>
  </result>
</xsl:template>
```

note: first author only (without xsl:for-each)
Apply templates

Recursive processing:

```xml
<xsl:template match=XPath >
  . . .
  <xsl:apply-templates select=XPath/>
  . . .
</xsl:template>
```

✓ Compare each selected child (descendant) of the matched source element against the templates in your program
✓ If a match is found, output the template for the matched node
✓ One can use `xsl:apply-templates` instead of `xsl:for-each`
✓ If the select attribute is missing, all the children are selected
✓ When the match attribute is missing, the template matches every node:
  ```xml
  <xsl:template> <xsl:apply-templates /> </xsl:template>
  ```
Q5: generate a HTML document consisting of the titles and authors of all books.

```xml
<xsl:template match="/">
  <html>
    <head> <title> Books </title> </head>
    <body>
      <ul>
        <xsl:apply-templates select="bib/book " /></ul></body>
  </html>
</xsl:template>

<xsl:template match="book">
  <li> <b> <xsl:value-of select="title ">, </xsl:value-of> </b>
    <xsl:for-each select="author "> <em><xsl:value-of /> </em> </xsl:for-each> <br>
  </li>
</xsl:template>
```
Query Languages for XML

- XPath
- XSLT
- XQuery
XQuery

W3C working draft [www.w3.org/TR/xquery](http://www.w3.org/TR/xquery)

Functional, strongly typed query language: Turing-complete

✓ XQuery = XPath + ...

  for LET-WHERE-RETURN (FLWR) ~ SQL’s SELECT-FROM-WHERE
  
  Sort-by

  XML construction (Transformation)

  Operators on types (Compile & run-time type tests)

+ User-defined functions

  Modularize large queries

  Process recursive data

+ Strong typing

  Enforced statically or dynamically
FLWR Expressions

For, Let, Where, OrderBy, return

Q1: Find titles and authors of all books published by Addison-Wesley after 1991.

<answer>{
    for $book in /bib/book
    where $book/@year > 1991 and $book/publisher='Addison-Wesley'
    return <book>
        <title> {$book/title } </title>,
        for $author in $book/author  return <author> {$author } </author>
    </book>
}</answer>

✓ for loop; $x: variable
✓ where: condition test; selection
✓ return: evaluate an expression and return its value
Find books that cost more at Amazon than at BN

<answer>{
    let $amazon := doc("http://www.amazon.com/books.xml"),
        $bn := doc("http://www.BN.com/books.xml")
    for $a in $amazon/books/book,
        $b in $bn/books/book
    where $a/isbn = $b/isbn and $a/price > $b/price
    return <book> {$a/title, $a/price, $b/price } </book>
}</answer>

✓ let clause
✓ join: of two documents
Conditional expression

Q2: Find all book titles, and prices where available
<answer>{
    for $book in /bib/book
    return <book>
        <title> {$book/title } </title>,
        { if $book[price]
            then <price> {$book/price } </price>
            else () }
    </book>
}</answer>
Indexing

Q3: for each book, find its title and its first two authors, and returns <et-al/> if there are more than two authors

<answer>
  for $book$ in /bib/book
  return <book>
    <title> {$book/title } </title>,
    { for $author$ in $book/author[position( ) <= 2]
      return <author> {$author } </author> }
    { if (count($book/author) > 2
      then <et-al/>
      else ( )
    }
  </book>
</answer>
Q4: find the titles of all books published by Addison-Wesley after 1991, and list them alphabetically.

<answer>{
  for $book in /bib/book
  where $book/@year > 1991 and $book/publisher='Addison-Wesley'
  order by $book/title
  return
    <book>
      <title> {$book/title } </title>,
      for $author in $book/author  return
        <author> {$author } </author>
    </book>
}</answer>
Q5: For each author, find titles of books he/she has written

<answer>{
    for $author in distinct(/bib/book/author)
    return <author name="{$author}" >{
        for $book in /bib/book
            where $book/author = $author
            return <title> {$book/title } </title>
    </author>
}</answer>

✓ Constructing attributes: <author name="{$author}" >
✓ Grouping: for $book in /bib/book ...
Recursion

Consider a part DTD

```xml
<!ELEMENT part (part*)>
<!ATTLIST part name CDATA #required>
<!ATTLIST part cost CDATA #required>
```

part – subpart hierarchy

Given a part element, we want to find the total cost of the part – recursive computation that descends the part hierarchy
function

```javascript
define function total (element part $part)
returns element part {
    let $subparts :=
        for $s in $part/part  return total($s)
    return {
        <part name="$part/@name"
            cost="$part/@cost + sum($subparts/@cost)">
        </part>
    }
}

✓ recursive function: it recursively descends the hierarchy of $part
✓ $subparts: a list
✓ $part: parameter
```
An XML document may come with an optional DTD – “schema”

```xml
<!DOCTYPE db [
  <!ELEMENT  db (book*)>
  <!ELEMENT  book  (title, chapter*, ref*)>
  <!ATTLIST book isbn ID #required>
  <!ELEMENT  chapter  (number, section*) >
  <!ELEMENT  section  (number, (text | section)*))>
  <!ELEMENT  ref  EMPTY>
  <!ATTLIST  ref to  IDREFS #implied>
  <!ELEMENT  title  #PCDATA>
  <!ELEMENT  text  #PCDATA>
]>
```
What is a DTD?

- A DTD constrains the structure of an XML document, and may help us formulate/optimize our queries.
- There is a relationship between a DTD and a databases schema or a type/class declaration of a program, but it is not close – hence the need for additional “typing” systems, such as XML Schema.
- A DTD is a syntactic specification. Its connection with any “conceptual” model may be quite remote.
- DTDs do not act like type systems for XQuery, XPath or XSLT. You can “validate” your XML documents, but that does not mean that your programs are checked for type errors.
For each element type $E$, a declaration of the form:

```xml
<!ELEMENT   E   P>
```

where $P$ is a regular expression, i.e.,

$$
P ::= \text{EMPTY} \mid \text{ANY} \mid \#PCDATA \mid E' \mid P_1, P_2 \mid P_1 \mid P_2 \mid P? \mid P+ \mid P^*\$$

- $E'$: element type
- $P_1, P_2$: concatenation
- $P_1 \mid P_2$: disjunction
- $P?$: optional
- $P+$: one or more occurrences
- $P^*$: the Kleene closure
Extended context free grammar: `<!ELEMENT E P>`

Why is it called extended?

E.g., `<!ELEMENT book (title, authors*, section*, ref*)>`

- single root: `<!DOCTYPE db [ ... ]>`
- subelements are ordered.

The following two definitions are different. Why?

`<!ELEMENT section (text | section)*>`

`<!ELEMENT section (text* | section* )>`

- recursive definition, e.g., section, binary tree:

  `<!ELEMENT node (leaf | (node, node))>`

  `<!ELEMENT leaf (#PCDATA)>`
Element Type Definition (3)

- more on recursive DTDs

```xml
<!ELEMENT  person (name, father, mother)>
<!ELEMENT  father  (person)>
<!ELEMENT  mother (person)>
```

What is the problem with this? How to fix it?
- Attributes
- optional (e.g., father?, mother?)
Element Type Definition (4)

✓ EMPTY element:
  <!ELEMENT ref EMPTY>
  <!ATTLIST ref to IDREFS #implied>
  observe that it has attributes

✓ ANY: may contain any content
  <!ELEMENT generic ANY>

✓ mixed content
  <!ELEMENT section (#PCDATA | section)>
Element Type Definition (5)

✓ global definition:

```
<!ELEMENT  person   (name,  ssn)>
<!ELEMENT  course   (name,  credit, instructor)>
```

The type definition associated with an element is unique -- only one declaration for name is allowed.

To avoid name clashes, one may use two distinct tags: e.g., personname, coursename.
Attribute declarations (1)

General syntax:

```
<!ATTLIST element_name
          attribute-name attribute-type default-declaration>
```

example: “keys” and “foreign keys”

```
<!ATTLIST book
          isbn   ID   #required>

<!ATTLIST ref
          to     IDREFS  #implied>
```

Note: it is OK for several element types to define an attribute of the same name, e.g.,

```
<!ATTLIST person  name  ID  #required>
<!ATTLIST pet      name  ID  #required>
```
<!ATTLIST element_name
    attribute-name attribute-type default-declaration>

✓ attribute types:
  – CDATA
  – ID, IDREF, IDREFS
  – ...

✓ default declarations:
  – #required, #implied
  – “default value”, #fixed “default value”
Specifying ID and IDREF attributes

<!ATTLIST
  person
  id       ID       #required
  father   IDREF    #implied
  mother   IDREF    #implied
  children IDREFS  #implied>

e.g.,

<person id="898" father="332" mother="336"
  children="982 984 986">

  ....
</person>
XML reference mechanism

✓ ID attribute: unique within the entire document.
  – An element can have at most one ID attribute.
  – No default (fixed default) value is allowed.
    • #required: a value must be provided
    • #implied: a value is optional

✓ IDREF attribute: its value must be some other element’s ID value already in the document.

✓ IDREFS attribute: its value is a set, each element of the set is the ID value of some other element in the document.

<person id="898" father="332" mother="336"
  children="982 984 986">
Keys and Foreign Keys

Example: school document

```xml
<!ELEMENT db   (student+, course+) >
<!ELEMENT student (id, name, gpa, taking*)>
<!ELEMENT course (cno, title, credit, taken_by*)>
<!ELEMENT taking   (cno)>
<!ELEMENT taken_by  (id)>
```

- **keys**: locating a specific object, an invariant connection from an object in the real world to its representation
  
  \[ \text{student.@id} \rightarrow \text{student}, \quad \text{course.@cno} \rightarrow \text{course} \]

- **foreign keys**: referencing an object from another object
  
  \[ \text{taking.@cno} \subseteq \text{course.@cno}, \quad \text{course.@cno} \rightarrow \text{course} \]
  
  \[ \text{taken_by.@id} \subseteq \text{student.@id}, \quad \text{student.@id} \rightarrow \text{student} \]
The limitations of ID/IDREF

ID and IDREF attributes in DTD vs. keys and foreign keys in RDBs

✓ Scoping:

- ID unique within the entire document (like oids), while a key needs only to uniquely identify a tuple within a relation
- IDREF untyped: one has no control over what it points to -- you point to something, but you don’t know what it is!

<student id="01" name="John Smith" taking="CS2"/>
@student id="02" name="Mary Brown" taking="CS2 01"/>
<course id="CS2"/>
The limitations of the XML standard (DTD)

- keys need to be multi-valued, while IDs must be single-valued (unary)

  enroll (sid: string, cid: string, grade:string)

- a relation may have multiple keys, while an element can have at most one ID (primary)

- ID/IDREF can only be defined in a DTD, while XML data may not come with a DTD/schema

- ID/IDREF, even relational keys/foreign keys, fail to capture the semantics of hierarchical data