XML & DTDs

csc343, Introduction to Databases
Bogdan Simion
(originally based on slides by Jeff Ullman and Diane Horton)
Fall 2015
Introduction

- The relational model is very rigid:
  - Everything must be a table.
  - The schema must be defined in advance.
  - Everything must conform to the schema.
- Relational DBMSs exploit this to give us data we can count on and efficient queries.
- But some data doesn’t fit the model well. For example, we may have
  - missing information, and
  - indeterminate quantities.
HTML to XML

• XML grew out of HTML, and is intentionally similar:
  • Tags and attributes
  • Tree-structured format

• But there are important differences:
  • XML data must be well-formed.
  • You define your own tags and attributes.
  • These describe the meaning of the data, and imply nothing about its presentation.
  • XML data is not meant for human consumption.
What’s XML for?

• XML is said to be “self-describing”.
  • Schema-like information is part of the data itself.
  • Example:
    `<student stnum="1234" name="Cindylou Who"`>
      `<address`>
        `<street>99 Alfalfa Way</street`
        `<city>Whoville</city`
      </address`
    </student>`
What’s XML for?

• XML is great for
  • Recording data that software needs.
  • Exchange of information between pieces of software.

• It is not a Database!
  • Never meant to be!
  • DBMSs have optimized execution, indexing, logging, ACID transactions, etc.
  • XML doesn’t scale (search, delete, insert, etc.)
  • High storage space (very verbose)
Well-formed vs valid XML

• Well-formed XML
  • Just need a single root element and proper nesting.
  • Any tag or attribute can go anywhere.

• Valid XML
  • A “DTD” (document type definition) specifies what tags and attributes are permitted, where they can go, and how many there must be.
  • A valid XML file is one that has a DTD and follows the rules specified in its DTD.
Well-formed XML

- Begin the document with a declaration, surrounded by `<?xml ... ?>`
- Declaration for a document that is merely well-formed (i.e., it has no DTD):
  `<?xml version="1.0" standalone="yes" ?>`
- The rest of the document is a single root tag with tags nested inside it.
Tags

- Tags can be matched pairs, leaving room for text or nested tags in between. Example:
  
  ```xml
  <tf-question qid="Q637" solution="False">
    <question>
      The Prime Minister, Justin Trudeau, is Canada's Head of State.
    </question>
  </tf-question>
  ```

- Or they be unmatched. Example:
  
  ```xml
  <response qid="Q637" answer="False" />
  ```
  
  Note the placement of the slash.

- Tag names are case-sensitive.
Example: quiz.xml
Attributes

- As we saw, an opening tag can have **attribute** name-value pairs within it. Example:
  ```html
  <tf-question qid="Q637" solution="False">
    <question>
      The Prime Minister, Justin Trudeau, is Canada's Head of State.
    </question>
  </tf-question>
  ```
- The pairs are separated by blanks.
- If all the information is in the attributes (no “sub-tags”), the tag becomes **Empty**.
We don’t need to use attributes

```xml
<tf-question qid="Q637" solution="False">
  <question>
    The Prime Minister...
  </question>
</tf-question>

could become:

```xml
<tf-question>
  <qid>Q637</qid>
  <solution>False</solution>
  <question>
    The Prime Minister...
  </question>
</tf-question>
```
The other extreme: all data via attributes

<tf-question qid="Q637" solution="False">
  <question>
    The Prime Minister ...
  </question>
</tf-question>

could become:

<tf-question qid="Q637" solution="False">
  question="The Prime Minister ...
  
</tf-question>
It’s a design decision

- In most cases, something in between makes more sense.
- Matched tags make sense when you need structure within.
- Attributes make sense when you want something like keys and foreign keys. (More on that later.)
Checking for well-formedness

- [http://validator.w3.org](http://validator.w3.org)
- `xmllint` command on cdf. Default is to check merely for well-formedness.
- `xmllint --debug`
  Outputs an annotated tree of the parsed document. Useful for diagnosis of problems.
XML documents have a tree structure

```xml
<?xml version="1.0" ?>
<!-- Some comment -->
<StudentId="111111111">
    <Name><First>John</First><Last>Doe</Last></Name>
    <Status>U2</Status>
    <CrsTaken CrsCode="CSC308" Semester="F1997" />
    <CrsTaken CrsCode="MAT123" Semester="F1997" />
</Student>
<StudentId="987654321">
    <Name><First>Bart</First><Last>Simpson</Last></Name>
    <Status>U4</Status>
    <CrsTaken CrsCode="CSC308" Semester="F1994" />
</Student>
</Students>
<!-- Some other comment -->
The document tree
Problems with merely well-formed XML

• There are no restrictions on
  • what tags are allowed
  • what order, nesting
  • what attributes each tag can have
  • what is mandatory and what is optional

• If a program is to process our XML, this would be very useful to know.
Valid XML with DTDs
Content of a DTD

• A series of rules.
• An **ELEMENT** rule defines an element that may occur, and what can be within its opening and closing tags.
• An **ATTLIST** rule defines an attribute of an element.
• Order of the rules doesn’t matter.
ELEMENT rules

• Form: `<!ELEMENT «name» ( «subcomponents» )>`

• **name**: the element’s tag.

• **subcomponents**: can be
  • A comma-separated list of elements. Meaning: the elements must occur inside, and in the order given.
  • `#PCDATA`
    Meaning: The element contains simply text (no subelements).
  • `EMPTY`
    Meaning: This is an “empty” element. It may have attributes, but not matching opening & closing tags.
Examples

<!ELEMENT INGREDIENT (NAME, QUANTITY)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT QUANTITY EMPTY>
More expressiveness for subcomponents

• We can use the pipe symbol | to indicate alternatives.

• We specify multiplicity as follows:
  • * means zero or more
  • + means one or more
  • ? means zero or one (i.e., the subcomponent is optional)

• We can use brackets for grouping.
ATTLIST rules

• Form:
  ```xml
  <!ATTLIST elName attName type optionality >
  ```
• `elName`: the element whose attribute this is.
• `attName`: the name of this attribute.
• `type`: either `CDATA` or a list of possible values, e.g., `True|False`.
• `optionality`: Either `#REQUIRED` or `#IMPLIED` (which means optional).

• You can define multiple attributes at once.
  ```xml
  <!ATTLIST person SIN CDATA #REQUIRED
  age CDATA #IMPLIED >
  ```
Example

<!ELEMENT RECIPES (RECIPE)+>
<!ELEMENT RECIPE (INGREDIENTS, STEPS)>
<!ATTLIST RECIPE name CDATA #REQUIRED>
<!ATTLIST RECIPE type CDATA #IMPLIED>
<!ATTLIST RECIPE keywords CDATA #IMPLIED>
<!ELEMENT INGREDIENTS (INGREDIENT)+>
<!ELEMENT INGREDIENT (NAME, QUANTITY)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT QUANTITY EMPTY>  
<!ATTLIST QUANTITY amount CDATA #REQUIRED>
<!ATTLIST QUANTITY units CDATA #IMPLIED>
<!ELEMENT STEPS (STEP+)>
<!ELEMENT STEP (#PCDATA)>
Using a DTD

• The declaration must say that the document is not standalone:
  
  ```xml
  <?xml version="1.0" standalone="no" ?>
  ```

• Three possible places for the DTD:
  
  • In the same file, between the declaration and the XML content.
  
  • In a separate file on the same computer. Specify the filename, or give the full or relative path.
  
  • At a URL.

• In all cases, you must specify what the root element will be.
DTD in the same file

<?xml version="1.0" standalone="no" ?>
<!DOCTYPE People [
  <!ELEMENT People (Person*)>
  <!ELEMENT Person (#PCDATA)>
]>

<People>
  <Person>Tommy Douglas</Person>
  <Person>Terry Fox</Person>
  <Person>Louise Arbour</Person>
  <Person>Chris Hadfield</Person>
</People>
DTD in another file

```xml
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE People SYSTEM "people.dtd">

<Person>Tommy Douglas</Person>
<Person>Terry Fox</Person>
<Person>Louise Arbour</Person>
<Person>Chris Hadfield</Person>

</People>
```
<xml version="1.0" standalone="no" ?>
<!DOCTYPE People SYSTEM "http://www.cs.utoronto.ca/~dianeh/xyyz/people.dtd">

<People>
  <Person>Tommy Douglas</Person>
  <Person>Terry Fox</Person>
  <Person>Louise Arbour</Person>
  <Person>Chris Hadfield</Person>
</People>
“Keys” and “foreign keys”
Motivation

• Just as in the relational model, we sometimes want
  • unique identifiers.
  • the ability to refer in one place to some data in another place.
• Example: quiz.xml
• We would like the DTD to express these rules and our tools to enforce them.
• DTDs don’t have this full capability, but they do have some modest features in this direction.
Using ID to enforce uniqueness

• To specify that values must be unique:
  • Make an attribute of type ID rather than CDATA.
  • Example:
    `<!ATTLIST mc-question qid ID #REQUIRED>`

• Values of ID attributes are restricted.
  • Must not begin with a digit.
  • Must not have blanks.
Limitations of ID

• Uniqueness is enforced across all IDs in the file.

• Example: In quiz.xml,
  • questions have an ID attribute called qid and
  • students have an ID attribute called sid.

• Implications:
  • If two questions have the same qid, or if two students have the same sid, is considered an error. ✓
  • If a question’s qid is the same as a student’s sid, this is considered an error. ✗
Using IDREF to enforce referential integrity

• To specify that a value must refer to some ID:
  • Make an attribute of type IDREF.
  • Example:
    `<!ATTLIST response qid IDREF #REQUIRED>`
  • We can allow an attribute to have a list of values, each of which references some ID:
    `<!ATTLIST response qid IDREFS #REQUIRED>`
Limitations of IDREF

• An IDREF attribute needs only to refer to any ID in the file, not specifically to one of a particular type.

• Example: In quiz.xml,
  • a response has a qid that is an IDREF.

• Implications:
  • If a response’s qid refers to nothing, this is considered an error. ✓
  • If a response’s qid refers to a student’s sid, this is considered fine. ✗
Checking for validity

- `xml lint --valid` command on cdf.
Limitations of DTDs

- ID and IDREF are a pale imitation of keys and foreign keys.
  - All ID values are treated as a single set.
- ID and IDREF only work within a single file.
  - References to an ID in another file are flagged as errors.
  - Duplicate ID values across files cannot be detected.
- There are no other types of constraints.
- The only data type is string.
- It is very inconvenient to specify contents but allow them in any order.
XML Schema

- XML Schema has greater expressive power.
  - Rich set of built-in types, plus user-defined types
  - Finer control over sequences of sub-elements.
  - More effective keys and foreign keys
- It is also much more complex.
- Note: XML Schema Definitions (XSDs) are themselves XML documents.
  - They describe “elements” and
  - the things doing the describing are themselves “elements”.