Semistructured data, XML, DTDs

Introduction to databases
CSCC43 Winter 2011
Ryan Johnson

Thanks to Ramona Truta and Renee Miller for material in these slides

Structured vs. unstructured data

• Databases are highly structured
  – Well-known data format: relations and tuples
  – Every tuple conforms to a known schema
  – Data independence? Woe unto you if you lose the schema

• Plain text is unstructured
  – Cannot assume any predefined format
  – Apparent organization makes no guarantees
  – Self-describing: little external knowledge needed
    ... but have to infer what the data means

Irony: database cannot stand alone

Motivation for self-describing data

• Consider a C struct
  • Description:
    • struct {
      int id;
      int type;
      char name[8];
      struct {
        double x;
        double y;
      } location;
    } shape;
  • Data at code level:
    • {1, 101, “square”,
      {1.5, 5.0}}
  • Data at byte-level:
    • 0x000000100000065
      0x7371756172650000
      0x3FF8000000000000
      0x4014000000000000
  • Pointers? Nightmare!

*Very* hard not to embed schema in logic

Enter semistructured data

• Observation: most data has some structure
  – Text: sentences, paragraphs, sections, ...
  – Books: chapters
  – Web pages: HTML

• Idea of semistructured data:
  – Enforce “well-formatted” data
    => Always know how to read/parse/manipulate it
  – Optionally, enforce “well-structured” data also
    => Might help us interpret the data, too

Pro: highly portable    Con: verbose/redundant
Why not use...

- HTML?
  ```html
  <dt style="color:red">id</dt><dd>1
  <dt>type</dt><dd>101
  <dt>name</dt><dd>square
  <dt>location</dt><dd>
    <dl>
      <dt>x</dt><dd>1.5
      <dt>y</dt><dd>5
    </dl>
  </dd>
  </div>
  ```
  - Pro: popular
  - Con: inconsistent, buggy
    - div, table, ul instead of dl?
    - Parsing is *hard*
  - Con: data+presentation
    - More like a query result
    - Fixed meaning for all tags

- JSON?
  ```json
  { 'id':1,  
    'type':101,  
    'name':'square',  
    'location':{    
      'x':1.5,  
      'y':5  
    }  
  }
  ```
  - Pro: portable
  - Con: underspecified
    - e.g. can’t constrain types
  - Con: schema still partly embedded
    - what do field names mean?

Features of XML

- Intentionally similar syntax to HTML
  - Tree-structured (hierarchical) format
  - Elements surrounded by opening and closing tags
  - Attributes embedded in opening tags
  - **Elements are nested**
  - Root element contains all others

- But with important differences
  - Strictly well-formed (must close all tags, etc.)
  - Tag/attribute names carry no semantic meaning
  - Data-only format: no implied presentation

- Valid identifiers (for elements and attributes)
  - [a-z][A-Z]

Both descendants of SGML

XML: designed for data interchange

```xml
<books search-terms="database+design">
  <book>
    <title>Database Design for Mere Mortals</title>
    <author>Michael J. Hernandez</author>
    <date>13/03/2003</date>
  </book>
  <book id="82">
    <title>Beginning Database Design</title>
    <subtitle>From Novice to Professional</subtitle>
    <author>Clare Churcher</author>
  </book>
</books>
```
XML terminology (cont.)

Valid names in XML

- Simple rules for elements/attributes names
  - may include letters (case sensitive)
  - may include (but not start with) digits and punctuation
  - no reserved words or keywords

- But lots of gotchas
  - Names must not start with “xml” (case insensitive)
  - Non-ASCII/latin letters: legal but not all parsers support them
  - Punctuation is iffy business (one exception: “.”)
    - Entity characters always verboten: < > & " '
    - Spec recommends “.” instead of “-” (real life: the opposite is true)
    - “.” is "reserved" for namespaces (not enforced)
    - “.” officially discouraged (real life: very rare)
    - “$” often used for parameter substitution by XML processors (XQuery, etc.)
    - Other punctuation vanishingly rare: @ # % ...
  - Upper case letters legal but fairly rare
    - All caps very rare (just like rest of Internet)
    - Often see "book-list" instead of camel case (e.g. BookList)

Rule of thumb: lower case and ‘-’ usually best

Rules for well-formed XML

- Must have a root element
- Every opening tag must have matching closing tag
- Elements must be properly nested
  - `<foo><bar></foo></bar>` is a no-no
- An attribute name can occur at most once in an opening tag. If it occurs,
  - It must have an explicitly specified value (Boolean attrs, like in HTML, are not allowed)
  - The value must be quoted (with ” or ‘)"
- XML processors not allowed to “fix” ill-formed documents (HTML browsers expected to!)

XML, text, and whitespace

- Adjacent non-tag chars parsed as “text” nodes
- Parser never ignores whitespace
  - Leading and trailing space left with its text node
  - Whitespace between tags produces “empty" text nodes

- Example:
  `<foo> hi<bar> ho </bar> </foo>`
Document type definition (DTD)

- Enforces more than “well-formed”-ness
  - Which entities may (or must) appear where
  - Attributes entities may (or must) have
  - Types attributes and data must adhere to
- DTD “separate” from XML it constrains
  - May be embedded in separate section
  - Most often referenced externally
- Validation: checking XML against its DTD(s)
  - Important for interpreting/validating data
  - Not necessary for parsing

Embedded vs. external DTD

- Specified as part of a document
  ```xml
  <?xml version="1.0" ?>
  <!DOCTYPE Book [
    ... ...
  ]>
  <Book> ... ... </Book>
  ```

- Reference to external (stand-alone) DTD
  ```xml
  <?xml version="1.0" ?>
  <Book> ... ... </Book>
  ```

DTD building blocks

- Elements (`<an-element>`...`</an-element>`)  
  - Must always close tags
  - If no contents: `<empty-element/>`
- Attributes (`<... an-attr="..."...>`)  
- Entities (“special” tokens)
  - e.g. `&lt;` `&gt;` `&amp;` `&apos;`
  - HTML defines lots of others (e.g. `&nbsp;`)
  - More on this later
- PCDATA (parsed character data)
  - Mixed text and markup
  - Use entities to escape '>', etc. which should not be parsed
- CDATA ([non-parsed] character data)
  - Plain text data
  - Tags not parsed, entities not expanded

DTD elements

- `<!ELEMENT $e ...>`
- May contain any of
  - Nothing: `<ELEMENT $e EMPTY>`
  - Anything: `<ELEMENT $e ANY>`
  - Text data: `<ELEMENT $ ($ (#PCDATA)>`
    - Always parsed [CDATA not allowed here]
  - Child elements: `<ELEMENT $e (...)>`
    - Any child referenced must also be declared
    - Child elements may themselves have children
  - Mixed content: `<ELEMENT $e (#PCDATA|...`
DTD elements: children

- Base construct: sequence
  ```
  <!ELEMENT $e (a)>
  <!ELEMENT $e (a, b, c, ...)>  
  - Children in XML must appear in DTD declaration order
- Either-or content
  ```
  <!ELEMENT $e (a|b|...)>  
  - Exactly one of the options must appear in the XML
- Constraining child cardinality
  ```
  <!ELEMENT $e (a, b+, c*, d?)>  
  - exactly one (a), at least one (b)
  - zero or more (c), at most one (d)
```

DTD attributes

- `<!ATTLIST $e $a $type $required>`
- Declares an attribute $a on element $e
- $type may be any of
  - character data: CDATA
  - one of a set of values: (v1|v2|...)  
  - unique identifier (or reference to one/many): ID[REF][S]
  - valid xml name (or list of names): NMTOKEN[S]
  - entity (or entities): ENTITY/ENTITIES
- $required may be
  - required (not required): #REQUIRED/#IMPLIED
  - fixed value (always the same): #FIXED "$value"
  - default value (used if none given): "$value"

DTD elements: examples

- `<!ELEMENT resume (bio,interests,education,experience,awards,service)>`
- `<!ELEMENT bio (name, addr, phone, email?, fax?, url?)>`
- `<!ELEMENT interests (interest+)>`
- `<!ELEMENT education (degree*)>`
- `<!ELEMENT awards ((award|honor)*)>`
- ...

Sequences and either-or can both nest

DTD attributes: examples

- `<!ATTLIST person
  sin ID #REQUIRED
  spouse IDREF #IMPLIED
  name CDATA "John Doe"
  trusted (yes|no) "no"
  species #FIXED "homo sapiens"
  alive (yes|no) #IMPLIED
  >`

#IMPLIED unless specified otherwise
**DTD attributes: ID[REF][S]**

- **ID attribute type**
  - Uniquely identifies an element in the document
  - Error to have two
  - Like HTML ‘id’ attribute, but can have any name
- **IDREF**
  - Refers to another element by ID
  - Error if corresponding ID does not exist
  - Like HTML ‘href’ attribute, but no ‘#’ needed
- **IDREFS**
  - List of IDREF attributes, separated by whitespace

*Problem: only one global set of IDs*

**DTD entities**

- **The XML equivalent of #define**
  ```xml
  <!ENTITY $name "$substituted-value">  
  ```
  - Can’t take parameters, though
- **Used just like other entities**
  ```xml
  <politician-speak>
  I vow to lead the fight to stamp out &buzz-word; by instituting powerful new programs that will ...
  </politician-speak>
  ```
  - Pick your favorite substitution:
    - `<ENTITY buzz-word “communism”>`
    - `<ENTITY buzz-word “racism”>`
    - `<ENTITY buzz-word “terrorism”>`
    - `<ENTITY buzz-word “illegal file sharing”>`

*Not heavily used: better templating methods exist*

**Limitations of DTDs**

- Don’t understand namespaces
- Very limited typing (just strings and xml names)
- Very weak referential integrity
  - All ID / IDREF / IDREFS use global ID space
- Can’t express unordered contents conveniently
  - How to specify that a,b,c must all appear, but in any order?
- All element names are global
  - Is `<name>` for people or companies?
  - can’t declare both in the same DTD

**XML Schema**

- Designed to improve on DTDs
- **Advantages:**
  - Integrated with namespaces
  - Many built-in types
  - User-defined types
  - Has local element names
  - Powerful key and referential constraints
- **Disadvantages:**
  - Unwieldy, much more complex than DTDs

*We won’t cover XML schema in class*