XML and Databases

Lecture 1
Introduction to XML

Sebastian Maneth
NICTA and UNSW

CSE@UNSW - Semester 1, 2009

XML

- Similar to HTML (Berners-Lee, CERN → W3C)
- use your own tags.
- Amount/popularity of XML data is growing steadily (faster than computing power)

Some of these challenges are

- Existing (DB) technology cannot be applied to XML data.
- can handle huge amounts of data stored in relations
  - storage management
  - index structures
  - join/sort algorithms
  - ...
This course will introduce you to the world of XML and to the challenges of dealing with XML in a RDMS.

Some of these challenges are:
- Existing (DB) technology cannot be applied to XML data.
- How do we store trees?
- Can we benefit from index structures?
- How to implement tree navigation?

Additional challenges posed by W3C’s XQuery proposal:
- A notion of order
- A complex type / schema system
- Possibility to construct new tree nodes on the fly.

**XML = Threat to Databases... ?!**

You will learn about:
- Tree structured data (XML)
- XML parsers & efficient memory representation
- Query languages for XML (XPath, XQuery, XSLT...)
- Efficient evaluation using finite-state automata
- Mapping XML to databases
- Advanced topics (query optimizations, access control, update languages...)

You will NOT learn about:
- Hacking CGI scripts
- HTML
- Java...
About XML

- We hope to convince you that XML is not yet another hyped TLA, but is useful technology.
- You will become best friends with one of the most important data structures in Computing Science, the tree. XML is all about tree-shaped data.
- You will learn to apply a number of closely related XML standards:
  > Representing data: XML itself, DTD, XMLSchema, XML dialects
  > Interfaces to connect PLs to XML: DOM, SAX
  > Languages to query/transform XML: XPath, XQuery, XSLT.

About XML

We will talk about algorithms and programming techniques to efficiently manipulate XML data:
- Regular expressions can be used to validate XML data
- Finite state automata lie at the heart of highly efficient XPath implementations
- Tree traversals may be used to preprocess XML trees in order to support XPath evaluation, to store XML trees in databases, etc.

In the end, you should be able to digest the thick pile of related W3C X_standard standards.
(Ike, XQuery, XPointer, XLink, XHTML, XInclude, XML Base, XML Schema, …)

Course Organization

Lecture
Thursday, 15:00 – 18:00
Central Lecture Block 4 (K-E19-G05)
Lecturer
Sebastian Maneth
Consult
Friday, 11:00-12:00 (E508, L5)
All email to
cs4317@cse.unsw.edu.au

Tutorial
Monday, 11:00-13:00 @ Quadrangle G045 (K-E15-G045)
Tuesday, 16:00-18:00 @ Quadrangle G028 (K-E15-G028)
Wednesday, 16:00-18:00 @ Quadrangle G048 (K-E15-G048)
Tutors
Andrew Clayphan, Kim Nguyen
All email to
cs4317@cse.unsw.edu.au

Programming Assignments
5 assignments, due every other Monday. (1st is due 29th March 2nd is due 6th April ... )
Per assignment: 12 points (total: 60 points)
Final exam: 40 points (must get 40% to pass!)

Outline - Lectures

1. Introduction to XML, Encodings, Parsers
2. Memory Representations for XML: Space vs Access Speed
3. RDBMS Representation of XML
4. DTDs, Schemas, Regular Expressions, Ambiguity
5. Node Selecting Queries: XPath
6. Efficient XPath Evaluation
7. XPath Properties: backward axes, containment test
8. Streaming Evaluation: how much memory do you need?
9. XPath Evaluation using RDBMS
10. XSLT
11. XSLT & XQuery
12. XQuery & Updates

Suggested reading material:
Course slides of Marc Scholl, Uni Konstanz
http://www.inf.uni-konstanz.de/dbs/teaching/ws0506/database-xml/XMLDB.pdf
http://arbre.is.s.u-tokyo.ac.jp/~hahosoya/xmlbook/

Book
None!

Course slides of Marc Scholl, Uni Konstanz
http://www.inf.uni-konstanz.de/dbs/teaching/ws0506/database-xml/XMLDB.pdf
Theory / PL oriented, book draft:
http://arbre.is.s.u-tokyo.ac.jp/~hahosoya/xmlbook/
Outline - Assignments

You can freely choose to program your assignments in

- C / C++, or
- Java

However, your code must compile with gcc / g++, javac, as installed on CSE Linux systems!!

Send your source code by Monday 23:59 (every other week) to

cs4317@cse.unsw.edu.au

Outline - Assignments

1. Read XML, using DOM parser. Create document statistics 12 days
2. SAX Parse into memory structure: Tree vs DAG 2 weeks
3. Map XML into RDBMS 1 week (+ break)
4. XPath evaluation over main memory structures (+ streaming support) 3 weeks
5. XPath into SQL Translation 2 weeks

Outline - Assignments

Hashing/hash code's (A2)

1. Read XML, using DOM parser. Create document statistics 12 days
2. SAX Parse into memory structure: Tree vs DAG 2 weeks
3. Map XML into RDBMS 1 week (+ break)
4. XPath evaluation over main memory structures (+ streaming support) 3 weeks
5. XPath into SQL Translation 2 weeks

Finite automata (A4)

Outline

1. Three motivations for XML
   - religious
   - practical
   - theoretical / mathematical
2. Well-formed XML
3. Character Encodings
4. Parsers for XML
   - parsing into DOM (Document Object Model)

XML Introduction

Religious motivation for XML:

to have one language to speak about data.
XML Motivation (religious cont.)

- XML is a Data Exchange Format

1974 SGML (Charles Goldfarb at IBM Research)
1989 HTML (Tim Berners-Lee at CERN/Geneva)
1994 Berners-Lee founds Web Consortium (W3C)
1996 XML (W3C draft, v1.0 in 1998)

Tom Henzinger
EPFL
Tom.Henzinger@epfl.ch

Helmut Seidl
TU Munich
seidl@inf.tum.de

(2) Practical

XML = data + structure

Is this a good "template"?? What about last/first name? Several affil's / email's...?
XML Documents

- Ordinary text files (UTF-8, UTF-16, UCS-4 ...)
- Originates from typesetting/DocProcessing community
- Idea of labeled brackets (“mark up”) for structure is not new! (already used by Chomsky in the 1960’s)
- Brackets describe a tree structure
- Allows applications from different vendors to exchange data!
- standardized, extremely widely accepted!

XML Documents

Problem
- highly verbose, lots of repetitive markup, large files

Contra...
- highly verbose, lots of repetitive markup, large files

Pro...
- we have a standard! A Standard! A STANDARD!
  - ☺ You never need to write a parser again! Use XML! ☺

... instead of writing a parser, you simply fix your own “XML dialect”,
by describing all “admissible templates” (+ maybe even the specific
data types that may appear inside).

You do this, using an XML Type definition language such as DTD or Relax NG (Oasis).

Of course, such type definition languages are SIMPLE, because you
want the parsers to be efficient!

They are similar to EBNF. ➔ context-free grammar with reg. expr’s in
the right-hand sides. ☺

XML Documents

Element names and their content

Example DTD (Document Type Description)

```
Related ➔ (colleague | friend | family)*
colleague ➔ (name,affil*,email*)
friend ➔ (name,affil*,email*)
family ➔ (name,affil*,email*)
name ➔ (#PCDATA)
```

Element names and their content
XML Documents

Example DTD (Document Type Description)

```
Related  → [colleague | friend | family]*
colleague → (name, affil*, email*)
friend    → (name, affil*, email*)
family    → (name, affil*, email*)
name      → (#PCDATA)
```

Element names and their content

```
Related
  →
  →
  →
friend  →
  →
colleague →
  →
family  →
  →
name    →
  →
```

Victor

What else: (besides element and text nodes)

- attributes
- processing instructions
- comments
- namespaces
- entity references (two kinds)
### XML Documents

What else:
- attributes
- processing instructions
- comments
- namespaces
- entity references (two kinds)

```xml
<family rel="brother",age="25">
  <name>
    ...
  </name>
</family>
```

See 2.6 Processing Instructions

```php
$sql("SELECT * FROM ...") ...?
```

<!-- the 'price' element's namespace is http://ecommerce.org/schema -->
```
<edi:price xmlns:edi='http://ecommerce.org/schema' units='Euro'>32.18</edi:price>
```

Type `<key>less-than</key>` (<`) to save options.

This document was prepared on &docdate; and

```html
<edi:price xmlns:edi='http://ecommerce.org/schema' units='Euro'>32.18</edi:price>
```
Early Markup

The term markup has been coined by the typesetting community, not by computer scientists.

With the advent of printing presses, writers and editors used (often marginal) notes to instruct printers to:
- Select certain fonts
- Let passages of text stand out
- Indent a line of text, etc

Proofreaders use a special set of symbols, their special markup language, to identify typos, formatting glitches, and similar erroneous fragments of text.

The markup language is designed to be easily recognizable in the actual flow of text.

Early Markup

Computer scientists adopted the markup idea – originally to annotate program source code:
- Design the markup language such that its constructs are easily recognizable by a machine.

Approaches
1. Markup is written using a special set of characters, disjoint from the set of characters that form the tokens of the program.
2. Markup occurs in places in the source file where program code may not appear (program layout).

Example: Fortran 77 fixed form source:
- Fortran statements start in column 7 and do not exceed column 72.
- A Fortran statement longer than 66 chars may be continued on the next line if a character not in \{ 0,!,_ \} is placed in column 6 of the continuing line.
- Comment lines start with a "C" or "*" in column 1.
- Numeric labels (DO, FORMAT statements) have to be placed in columns 1-5.

Sample Markup Application:

A Comic Strip Finder
- Next 8 slides from Marc Scholl’s 2005 lecture.
Stage 2: HTML-Style Physical Markup

```html
dilbert.html
<br>
<ti>Speed is the key to success.</ti>
<br>
<ti>Is it okay to do things wrong if we're really really fast?</ti>
<br>
<ti>Wally</ti> <br>How I'm all confused.
```

Stage 3: XML-Style Logical Markup

- We create a set of tags that is customized to represent the content of comics, e.g.:

  ```xml
  <character>Dilbert</character>
  <bubble>Speed is the key to success.</bubble>
  ```

- New types of queries may require new tags: No problem for XML!
  - Resulting set of tags forms a new markup language (XML dialect).
- All tags need to appear in properly nested pairs (e.g., `<b>Hello</b>`).
- Tags can be freely nested to reflect the logical structure of the comic content.

### Parsing XML?

In comparison to the stage 1 ASCII-level markup parsing, how difficult do you rate the construction of an XML parser?

Stage 4: Full-Featured XML Markup

- Although fairly simplistic, the previous stage clearly constitutes an improvement.
- XML comes with a number of additional constructs which allow us to convey even more useful information, e.g.:
  - Attributes may be used to qualify tags (avoid the so-called tag soup).
  - References establish links internal to an XML document:

```xml
<character id="pff">The Pointy-Haired Boss</character>
```

### In our example

```xml
dilbert.xml
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
<br>
```
Today, XML has many friends:

**Query Languages**
- XPath, XSLT, XQuery, fxt... (mostly by W3C)

**Implementations**
- (Parsers, Validators, Translators)
  - SAX, Xalan, Galax, Xerxes, ...
  - (by IBM/Apache, Microsoft, Oracle, Sun...)

**Current Issues**
- DB/PL support ("data binding", JBind, Castor, Zeus...)
- storage support (compression, data optimization)

---

**XML, typical usage scenario**

```
<Product>
  <product_id> m101 </product_id>
  <name> Sony walkman </name>
  <currency> AUD </currency>
  <price> 200.00 </price>
  <gst> 10% </gst>
</Product>
```

---

**(3) Theoretical / Mathematical**

**Regular Tree languages (REGT).**

- Many characterizations:
  - Reg. Tree Grammars
  - Tree Automata
  - MSO Logic

- Nice properties:
  - Closed under intersection (union, complement)
  - Decidable equivalence

- CF: \( a^*b/c \)
- REGT (as expressions): \( a^*b/c^* \)

---

**2. Well-Formed XML**

From the W3C XML recommendation

```
http://www.w3.org/TR/REC-xml/
```

A textual object is well-formed XML if,

1. taken as a whole, it matches the production labeled \( \text{document} \)
2. it meets all the well-formedness constraints given in this specification...

\( \text{document} = \) start symbol of a context-free grammar ("XML grammar")

- \( (1) \) contains the context-free properties of well-formed XML
- \( (2) \) contains the context-dependent properties of well-formed XML

There are 10 WFCs (well-formedness constraints).

E.g.: **Element Type Match**

- "The Name in an element’s end tag must match the element name in the start tag."

---

**2. Well-Formed XML**

Context-free grammar in EBNF = System of production rules of the form

```
lhs ::= rhs
```

- \( \text{lhs} \) a nonterminal symbol (e.g., \( \text{document} \))
- \( \text{rhs} \) a string over nonterminal and terminal symbols.

Additionally (EBNF), we may use regular expressions in \( \text{rhs} \).

Such as:

- \( r^* \) zero or more repititions
- \( r+ \) one or more repititions
- \( r? \) optional \( r \)
- \( [abc] \) character class
As usual, the XML grammar can be systematically transformed into a program, an XML parser, to be used to check the syntax of XML input.

Parsing XML

1. Starting with the symbol `document`, the parser uses the `lhs::=rhs` rules to expand symbols, constructing a parse tree.
2. Leaves of the parse tree are characters which have no further expansion
3. The XML input is parsed successfully if it perfectly matches the parse tree’s front (concatenate the parse tree’s leaves from left-to-right, while removing `ε` symbols).

Example 1
Parse tree for XML input

```
<bubble speaker="phb">Um... No.</bubble>
```

Example 2
Parse tree for XML input

```
<?xml version="1.0"?><foo/>
```

2. Well-Formed XML

Terminology

- tags `name`, `email`, `author`, ...
- start tag `<name>`, `end tag` `</name>`
- elements `<name>`, `</name>`, `<author>`, `</author>`
- elements may be nested
- empty element `<red></red>` abbrev. `<red>`
- an XML document: single root element
  `<someTag> ... </someTag>`

- well-formed constraints
  - begin/end tags match
  - no attribute name may appear more than once
    - in a start tag or empty element tag
  - a parsed entity must not contain a recursive reference to itself, either directly or indirectly

Well-formed XML (fragments)

```
<Staff>
  <Name>
    <FirstName>Sebastian</FirstName>
    <LastName>Mann</LastName>
  </Name>
  <Login>smaneth</Login>
  <Ext>2481</Ext>
</Staff>
```

Non-well-formed XML

```
<foo> oops </bar>
<foo> oops </foo>
<a> oops </a>
```

```xml
<s0>
  <a b=c d>e</a>
</s0>
```
Well-formed XML (fragments)

```xml
<Staff>
  <Name>
    <FirstName> Sebastian </FirstName>
    <LastName> Maneth </LastName>
  </Name>
  <Login> smaneth </Login>
  <Ext> 2481 </Ext>
</Staff>
```

Questions

How can you implement the three well-formed constraints?

When, during parsing, do you apply the checks?

Non-well-formed XML

```xml
<foo> oops </bar>
<foo> oops </Foo>
<foo> oops .. <EOT>
<staff>
  <name> <firstname> <lastname>
  <login> smaneth </login>
  <ext> 2481 </ext>
</staff>
```

Character Encoding

- For a computer, a character like X is nothing but an 8 (16/32) bit number whose value is interpreted as the character X, when needed.
- Problem: many such number → character mappings, the so called encodings are in use today.
- Due to the huge amount of characters needed by the global computing community today (Latin, Hebrew, Arabic, Greek, Japanese, Chinese ...), conflicting intersections between encodings are common.

Example

| 0xcb 0xe4 0xd3 | iso-8859-7 | δ Σ |
| 0xcb 0xe4 0xd3 | iso-8859-15 | E a O |

Unicode

- The Unicode [http://www.unicode.org](http://www.unicode.org) Initiative aims to define a new encoding that tries to embrace all character needs.
- The Unicode encoding contains characters of “all” languages of the world plus scientific, mathematical, technical, box drawing, ... symbols

Range of the Unicode encoding: 0x0000-0x10FFFF (≈16*65536)

- Codes that fit into the first 16 bits (denoted U+0000-U+FFFF) encode the most widely used languages and their characters (Basic Multilingual Plane, BMP)
- Codes U+0000-U+007F have been assigned to match the 7-bit ASCII encoding which is pervasive today.

Unicode Transformation Formats

Current CPUs operate most efficiently on 32-bit words (16-bit words, bytes)

Unicode thus developed Unicode Transformation Formats (UTFs) which define how a Unicode character code between U+0000 and U+10FFFF is to be mapped into a 32-bit word (16-bit word, byte).

UTF-32

- Simply map exactly to the corresponding 32-bit value
- For each Unicode character in UTF-32: waste of at least 11 bits!

UTF-16

- Map a Unicode character into one or two 16-bit words
- U+0000 to U+FFFF map exactly to the corresponding 16-bit value
- above U+FFFF: subtract 0x10000 and then fill the □’s in

\[
\begin{align*}
1101 1000 & \text{ xxxx xxxx} \\
1101 1001 & \text{ xxxx xxxx}
\end{align*}
\]

E.g. Unicode character 0x012345

\[
(0x012345 - 0x10000 = 0x02345)
\]

UTF-16: 1101 1000 0000 0000 1101 1111 0100 0101

Note

UTF-16 works correctly, because the character codes between

1101 1000 0000 0000 and

1101 1111 1010 1011 (with each □ replaced by a 0)

are left unassigned in Unicode!!!!
UTF-8
Maps a unicode character into 1, 2, 3, or 4 bytes.

<table>
<thead>
<tr>
<th>Unicode range</th>
<th>Byte sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+000000</td>
<td>01000000</td>
</tr>
<tr>
<td>U+00007F</td>
<td>01000000</td>
</tr>
<tr>
<td>U+0007FF</td>
<td>11001000 10101000</td>
</tr>
<tr>
<td>U+010000</td>
<td>11001000 10101000 10101000</td>
</tr>
</tbody>
</table>

Spare bits (□) are filled from right to left. Pad to the left with 0-bits.

E.g. U+00A9 in UTF-8 is 11000010 10101001
U+2260 in UTF-8 is 11100010 10001001 10100000

UTF-8
→ For a UTF-8 multi-byte sequence, the length of the sequence is equal to the number of leading 1-bits (in the first byte)
→ Character boundaries are simple to detect
→ UTF-8 encoding does not affect (binary) sort order
→ Text processing software designed to deal with 7-bit ASCII remains functional.
   (especially true for the C programming language and its string [char[]] representation)

XML and Unicode
→ A conforming XML parser is required to correctly process UTF-8 and UTF-16 encoded documents. (The W3C XML Recommendation predates the UTF-32 definition)
→ Documents that use a different encoding must announce so using the XML text declaration, e.g.,
   `<?xml encoding="iso-8859-15"?>`
   or  `<?xml encoding="utf-32"?>`
→ Otherwise, an XML parser is encouraged to guess the encoding while reading the first bytes of the input XML document:

<table>
<thead>
<tr>
<th>Head of doc (bytes)</th>
<th>Encoding guess</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 0x3C 0x00 0x3F</td>
<td>UTF-16 (Little Endian)</td>
</tr>
<tr>
<td>0x3C 0x00 0x3F 0x00</td>
<td>UTF-16 (Big Endian)</td>
</tr>
<tr>
<td>0x3C 0x3F 0x78 0x6D</td>
<td>UTF-8 (or ASCII)</td>
</tr>
</tbody>
</table>

Notice: < = U+003C, ? = U+003F, x = U+0078, m = U+006D

The XML Processing Model
→ On the physical side, XML defines nothing but a flat text format, i.e., it defines a set of (e.g. UTF-8/16) character sequences being well-formed XML.
→ Applications that want to analyze and transform XML data in any meaningful way will find processing flat character sequences hard and inefficient!
→ The nesting of XML elements and attributes, however, defines a logical tree-like structure.

<table>
<thead>
<tr>
<th>Staff</th>
<th>Name</th>
<th>Login</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FirstName</td>
<td></td>
<td>LastName</td>
</tr>
</tbody>
</table>
**The XML Processing Model**

- Virtually all XML applications operate on the logical tree view which is provided to them by an XML processor (i.e., "parse & store").
- XML processors are widely available (e.g., Apache's Xerces).

How is the XML processor supposed to communicate the XML tree structure to the application?

- For many PL's there are "data binding" tools.
  Gives very flexible way to get PL view of the XML tree structure.

But first, let's see what the standard says…

### Accessor Functions ("node properties")

<table>
<thead>
<tr>
<th>Node type</th>
<th>Property</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOC</td>
<td>children: DOC</td>
<td>ELEM</td>
</tr>
<tr>
<td></td>
<td>base-uri: DOC-STRING</td>
<td>version: DOC-STRING</td>
</tr>
<tr>
<td>ELEM</td>
<td>local-name: ELEM-STRING</td>
<td>version: ELEM-STRING</td>
</tr>
<tr>
<td></td>
<td>children: ELEM</td>
<td>attributes: ELEM-ATTR</td>
</tr>
<tr>
<td></td>
<td>parent: ELEM</td>
<td></td>
</tr>
<tr>
<td>ATTR</td>
<td>local-name: ATTR-STRING</td>
<td>value: ATTR-STRING</td>
</tr>
<tr>
<td></td>
<td>owner: ATTR-ELEMENT</td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td>code: CHAR-UNICODE</td>
<td>parent: CHAR-ELEMENT</td>
</tr>
</tbody>
</table>

XML Information Set: [http://www.w3.org/TR/xml-infoset](http://www.w3.org/TR/xml-infoset)

### Questions

1. A NODE type can be one of DOC, ELEM, ATTR, or CHAR.
   - In the two places of the property functions where NODE appears, which of the four types may actually appear there?
   - For instance, is this allowed?
     ```
     local-name(e1) = "forecast"
     children(e1)  = [c1,e2,c2,c3,c4,c5]
     ```

2. Are there property functions that are "redundant"? (meaning they can be computed from other property functions already)
   - Which sets of property functions are "minimal"?

3. What about WHITESPACE?
   - Where in an XML document does it matter, and where not?
   - Where in the Infoset are the returns and indentations of the document?
   - (did we do a mistake? If so, what is the correct Infoset?)
Querying the Infoset

Using the Infoset, we can analyse a given XML document in many ways. For instance:

→ Find all ELEM nodes with localname=bubble, owning an ATTR node with localname=speaker and value=Dilbert.

→ List all scene ELEM nodes containing a bubble spoken by "Dogbert".

→ Starting in panel 2 (ATTR no), find all bubbles following those spoken by "Alice".

Such queries appear very often and can conveniently be described using XPath queries:

Æ //bubble[@speaker="Dilbert"]/.. //scene
Æ //bubble[@speaker="Dogbert"]/.. //panel[@no="2"]/bubble
Æ //panel[@no="2"]//bubble[@speaker="Alice"]/following::bubble

3. Parsers for XML

Two different approaches:

(1) Parser stores document into a fixed (standard) data structure (e.g., an Infoset compliant data structure, such as DOM)

→ W3C standard, see http://www.w3.org/TR/REC-DOM-Level-1/

(2) Parser triggers "events". Does not store!

User has to write own code on how to store / process the events triggered by the parser.

DOM – Document Object Model

→ Language and platform-independent view of XML

→ DOM APIs exist for many PLs (Java, C++, C, Perl, Python, …)

DOM relies on two main concepts

(1) The XML processor constructs the complete XML document tree (in-memory)

(2) The XML application issues DOM library calls to explore and manipulate the XML tree, or to generate new XML trees.

Advantages
• easy to use
• once in memory, no tricky issues with XML syntax anymore
• all DOM trees serialize to well-formed XML (even after arbitrary updates)!

Disadvantage
Uses LOTS of memory!!

DOM Level 1 (Core)

NameNodeMap Node NodeList ProcessingInstruction CharacterData Attr Element Document

Text Comment CDATASection

Character strings (DOM type DOMString) are defined to be encoded using UTF-16 (e.g., Java DOM represents type DOMString using its String type).
**DOM Level 1 (Core)**

Name, Value, and attributes depend on the type of the current node.

<table>
<thead>
<tr>
<th>nodeType</th>
<th>nodeValue</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>nodeName</td>
<td>a NamedNodeMap</td>
</tr>
<tr>
<td>Attr</td>
<td>name</td>
<td>value of attribute</td>
</tr>
<tr>
<td>Text</td>
<td>content</td>
<td>content of the text node</td>
</tr>
<tr>
<td>CDATASection</td>
<td>CData-sect</td>
<td>content of the CDATA Section</td>
</tr>
<tr>
<td>EntityReference</td>
<td>entityReference</td>
<td>content of the entity reference</td>
</tr>
<tr>
<td>Entity</td>
<td>entityName</td>
<td>entity value</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>target</td>
<td>target context excluding the target</td>
</tr>
<tr>
<td>Comment</td>
<td>comment</td>
<td>content of the comment</td>
</tr>
<tr>
<td>Document</td>
<td>doctype</td>
<td>document type name</td>
</tr>
<tr>
<td>DocumentType</td>
<td>doctypeType</td>
<td>document type</td>
</tr>
<tr>
<td>DocumentFragment</td>
<td>fragment</td>
<td>document fragment</td>
</tr>
<tr>
<td>Notation</td>
<td>notationName</td>
<td>notation</td>
</tr>
</tbody>
</table>

**DOM Level 1 (Core)**

Some details

Creating an element/attribute using `createElement`/`createAttribute` does not wire the new node with the XML tree structure yet.

- Call `insertBefore`, `replaceChild`, …, to wire a node at an explicitly position

DOM type `NodeList` makes up for the lack of collection data types in most programming languages

DOM type `NamedNodeMap` represents an association table (nodes may be accessed by name)

**Example:**

```xml
<bubble>
  <getAttributes>
    "speaker" → "Dogbert"
  </getAttributes>
</bubble>
```

Methods: `getNamedItem`, `setNamedItem`, …

**Questions**

Given an XML file of, say, 50K, how large will be its DOM representation in main memory?

How much larger, in the worst case, will a DOM representation be with respect to the size of the XML document?

(difficult!)

How could we decrease the memory need of DOM, while preserving its functionality?

---

**END Lecture 1**

Next

"Even trigger" Parsers for XML:

- Build your own XML data structure and fill it up as the parser triggers input "events".