CMSC 724: XML

Amol Deshpande

University of Maryland, College Park

April 26, 2012
XML

- eXtensible Markup Language
  - Came out of document community
  - Simplified subset of: Standard Generalized Markup Language (SGML)
- De facto data exchange format
  - Self-describing (although beware of Semantic Heterogeneity)
  - Text (passes through firewalls, compresses well)

NOTE: Somewhat older paper. Different languages popular today. Much work since then on this topic. Also: JSON appears to be overtaking XML.
XML

- eXtensible Markup Language
  - Came out of document community
  - Simplified subset of: Standard Generalized Markup Language (SGML)
- De facto data exchange format
  - Self-describing (although beware of Semantic Heterogeniety)
  - Text (passes through firewalls, compresses well)

**NOTE:**
- Somewhat older paper
  - Different languages popular today
    - XPath, XQuery etc..

- Much work since then on this topic
- Also: JSON appears to be overtaking XML
XML: Example

- From the Irisnet project...
Example XML: Parking Space Information

```xml
<State id="Pennsylvania">
    <County id="Allegheny">
        <City id="Pittsburgh">
            <Neighborhood id="Oakland">
                <total-spaces>200</total-spaces>
                <Block id="1">
                    <GPS>…</GPS>
                    <pSpace id="1">
                        <in-use>no</in-use>
                        <metered>yes</metered>
                    </pSpace>
                    <pSpace id="2">
                        ...
                    </pSpace>
                </Block>
            </Neighborhood>
        </City>
    </County>
</State>
```

[[pSpace == parking space ]]
Example XML Fragment for PSF

- State id='Pennsylvania'
- County id='Allegheny'
- City id='Pittsburgh'
- Neighborhood id = 'Oakland'
- Neighborhood id = 'Shadyside'
- Block id = '1'
- Block id = '2'
- pSpace id = '1'
- pSpace id = '2'
- in-use
  - no
  - yes
- Price
- metered
- GPS
- total-spaces
- 200
XML Standardization

- XML may allow arbitrary structures, but need **schemas** and **namespaces** to exchange data.
- Schema languages
  - Initially DTD (Document Type Definition)
  - XMLSchema is more standard now
  - However XMLSchema is considered too complex, and there are many alternatives
    - RELAX NG
    - Schematron, Examplotron etc...
    - See for a comparison
- Purpose of namespaces is to mainly avoid duplicate attribute/element names
  - But also commonly used to define attributes or elements
  - See here for more information
XML DTDs

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE people_list [  
  <!ELEMENT people_list (person)>  
  <!ELEMENT person (name, birthdate?, gender?, socialsecuritynumber?)>  
  <!ELEMENT name (#PCDATA)>  
  <!ELEMENT birthdate (#PCDATA)>  
  <!ELEMENT gender (#PCDATA)>  
  <!ELEMENT socialsecuritynumber (#PCDATA)>  
]

<people_list>
  <person>
    <name>Fred Bloggs</name>
    <birthdate>2008-11-27</birthdate>
    <gender>Male</gender>
  </person>
</people_list>
```
XML Schema

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Address">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Recipient" type="xs:string" />
        <xs:element name="House" type="xs:string" />
        <xs:element name="Street" type="xs:string" />
        <xs:element name="Town" type="xs:string" />
        <xs:element name="County" type="xs:string" minOccurs="0" />
        <xs:element name="PostCode" type="xs:string" />
        <xs:element name="Country">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="FR" />
              <xs:enumeration value="DE" />
              <xs:enumeration value="ES" />
              <xs:enumeration value="UK" />
              <xs:enumeration value="US" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
XML Schema

```xml
<?xml version="1.0" encoding="utf-8"?>
<Address xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="SimpleAddress.xsd">
    <Recipient>Mr. Walter C. Brown</Recipient>
    <House>49</House>
    <Street>Featherstone Street</Street>
    <Town>LONDON</Town>
    <PostCode>EC1Y 8SY</PostCode>
    <Country>UK</Country>
</Address>
```
XML Query Languages

- XPath: Identify a set of nodes in the document
  - Used by both XSLT and XQuery to enumerate/identify nodes
- XSLT: Transformation language
  - Fairly verbose... essentially a program that traverses the document
  - "... whose primary goal was to render XML for the human reader on screen"
- XQuery: The current standard
  - Personally, I think it is too complicated
  - Likely only a subset will be used/implemented in practice
Example Queries

- Users issue queries against the document as a whole
  - Find all available parking spots in Oakland
    
    ```xml
    /State[@id="Pennsylvania"]/County[@id="Allegheny”]/City[@id="Pittsburgh”]
    /Neighborhood[@id="Oakland”]/Block/pSpace[in-use = “no”]
    ```

  - Find all blocks in Allegheny have more than 20 metered parking spots
    
    ```xml
    /State[@id="Pennsylvania”]/County[@id="Allegheny”]
    //Block[count(.//pSpace[metered = “yes”]) > 20]
    ```

  - Find the cheapest parking spot in Oakland Block 1
    
    ```xml
    /State[@id="Pennsylvania”]/County[@id="Allegheny”]/City[@id="Pittsburgh”]
    /Neighborhood[@id="Oakland”]/Block[@id='1’]
    /pSpace[not(../pSpace/price > ./price)]
    ```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml">

    <xsl:output method="xml" indent="yes" encoding="UTF-8"/>

    <xsl:template match="/persons">
        <html>
            <head> <title>Testing XML Example</title> </head>
            <body>
                <h1>Persons</h1>
                <ul>
                    <xsl:apply-templates select="person">
                        <xsl:sort select="family-name" />
                    </xsl:apply-templates>
                </ul>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="person">
        <li>
            <xsl:value-of select="family-name"/> <xsl:text>, </xsl:text>
            <xsl:value-of select="name"/>
        </li>
    </xsl:template>
</xsl:stylesheet>
The sample XQuery code below lists the unique speakers in each act of Shakespeare's play Hamlet, encoded in `hamlet.xml`:

```
<html><head/>
<body>
{
    for $act in doc("hamlet.xml")/ACT
    let $speakers := distinct-values($act/SPEAKER)
    return
    <div>
        <h1>{ string($act/TITLE) }</h1>
        <ul>
        { for $speaker in $speakers
            return <li>{ $speaker }<li>
        }
    </ul>
    </div>
}
</body></html>
```
XML Storage

- Option 1: Using a *native XML database*
  - Special purpose data stores
  - Going back to hierarchical/network models?
  - Many developed over the years, some by big names: see e.g., PureXML by IBM
  - Disadvantages: performance; need to re-build transaction/concurrency support etc
    - Often XML constructed from relational for data exchange
    - Often need support to do relational query processing (e.g., OLAP) on XML data
XML Storage

- Option 1: Using a *native XML database*
  - Special purpose data stores
  - Going back to hierarchical/network models?
  - Many developed over the years, some by big names: see e.g., PureXML by IBM
  - Disadvantages: performance; need to re-build transaction/concurrency support etc
    - Often XML constructed from relational for data exchange
    - Often need support to do relational query processing (e.g., OLAP) on XML data

- Option 2: Using relational databases
  - May lose structure in the XML document
  - Need to develop schemes to convert back and forth
  - XML queries naturally hierarchical → need many joins
XML Storage in Oracle

- From: Oracle XML DB
- Options:
  - Structured: Stored relationally – best performance
  - Binary XML storage: post-parsed binary format
    - Compact, post-parsed, XML Schema-aware
  - Unstructured: Stored in Character Large Object (CLOB)
    - Limited cases, e.g., when "document fidelity" needed
    - Mostly full document retrieval
- The white paper discusses in detail when to use which storage option
### XML Storage in Oracle

<table>
<thead>
<tr>
<th>QUALITY</th>
<th>STRUCTURED STORAGE</th>
<th>BINARY XML STORAGE</th>
<th>UNSTRUCTURED (CLOB) STORAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>XML decomposition can result in reduced throughput when ingesting or retrieving the entire content of an XML document.</td>
<td>+ High throughput. Fast DOM loading. There is a slight overhead from the binary encoder / decoder.</td>
<td>++ High throughput when ingesting and retrieving the entire content of an XML document.</td>
</tr>
<tr>
<td>Indexing support</td>
<td>++ B-tree, Bitmap, Oracle Text, XMLIndex, and function-based indexes.</td>
<td>+ XMLIndex, function-based, and Oracle Text indexes.</td>
<td>+ XMLIndex, function-based, and Oracle Text indexes.</td>
</tr>
<tr>
<td>Queries</td>
<td>++ Extremely Fast. Relational query performance. Users can create B-tree indexes on the exploded columns.</td>
<td>+. Fast when using XMLIndex. User queries which cannot use the index use streaming Xpath evaluation, which is reasonably fast as well.</td>
<td>--. Fast when using XMLIndex. Parts of query which can't use the index cannot be optimized.</td>
</tr>
<tr>
<td>Update operations (DML)</td>
<td>++ Extremely fast. Relational column gets updated in-place.</td>
<td>+ In-place, piecewise update for SecureFile LOB storage.</td>
<td>-- When any part of the document is updated, the entire document must be written back to disk.</td>
</tr>
</tbody>
</table>
## XML Storage in Oracle

<table>
<thead>
<tr>
<th>Space efficiency (disk)</th>
<th>++ Extremely space-efficient.</th>
<th>+ Space-efficient.</th>
<th>– Consumes the most disk space, due to insignificant white space and repeated tags.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data flexibility</td>
<td>– Limited flexibility. Only documents that conform to the XML schema can be stored in the XMLType table or column.</td>
<td>+ Flexibility in the structure of the XML documents that can be stored in an XMLType column or table.</td>
<td>+ Flexibility in the structure of the XML documents that can be stored in an XMLType column or table.</td>
</tr>
<tr>
<td>XML schema flexibility</td>
<td>– One XMLType table can only store documents conforming to one schema. Also provides relational-like in-place schema evolution capability.</td>
<td>++ Can store schemaless or schema based documents. An XMLType table can store documents conforming to any of the registered schemas.</td>
<td>++ Can store schemaless or schema based documents. Cannot use multiple XML schemas for the same XMLType table.</td>
</tr>
<tr>
<td>XML fidelity</td>
<td>+ DOM fidelity: A DOM created from an XML document that has been stored in the database will be</td>
<td>+ DOM fidelity (see structured storage description).</td>
<td>++ Document fidelity: Maintains the original XML data, byte for byte. In particular, all original white space is</td>
</tr>
</tbody>
</table>
## XML Storage in Oracle

<table>
<thead>
<tr>
<th>QUALITY</th>
<th>STRUCTURED STORAGE</th>
<th>BINARY XML STORAGE</th>
<th>UNSTRUCTURED (CLOB) STORAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized memory management</td>
<td>+ XML operations can be optimized to reduce memory requirements.</td>
<td>+ XML operations can be optimized to reduce memory requirements.</td>
<td>− XML operations on the document require creating a DOM from the document.</td>
</tr>
<tr>
<td>Validation upon insert</td>
<td>+ XML data is partially validated when it is inserted.</td>
<td>++ XML schema-based data can be fully validated when it is inserted, though this is an expensive operation.</td>
<td>+ XML schema-based data is partially validated when it is inserted.</td>
</tr>
<tr>
<td>Partitioning</td>
<td>++ Available</td>
<td>+ Partition based on virtual columns.</td>
<td>+ XMLType columns can be partitioned when the partitioning key is a relational column.</td>
</tr>
<tr>
<td>Streams based replication</td>
<td>- Not available</td>
<td>- Not available</td>
<td>++ Available</td>
</tr>
<tr>
<td>Compression and Encryption</td>
<td>Each element/attribute can be compressed / encrypted individually</td>
<td>Binary XML with SecureFile storage can be compressed / encrypted</td>
<td>Cannot be compressed / encrypted.</td>
</tr>
</tbody>
</table>
XML in RDBMS

- NOTE: Somewhat older paper... much work since then
- Key issues
  - Converting an XML document to relational
    - Called “shredding”
    - Uses the DTD Information
  - Processing queries
  - Converting the relational data back to XML
    - Essentially a query + some post-processing (maybe as a UDF)
XML in RDBMS

1. Simplify the DTD
   - The conversion can be a one-way process
   - No need to preserve exact structure in the relational schema
   - Order is important in XML
     - See a later paper: Handling order when converting
XML in RDBMS

1. Simplify the DTD
   - The conversion can be a one-way process
   - No need to preserve exact structure in the relational schema
   - Order is important in XML
     - See a later paper "Handling order when converting"

2. Create a set of tables
   - Simple option: Create a table for each element
     - Too many tables; a lot of joins needed later
     - Can think of that as denormalizing
   - Should try *inlining* as much as possible
XML in RDBMS: Example

**Figure 1**

```xml
<book>
  <booktitle>The Selfish Gene</booktitle>
  <author id="dawkins">
    <name>
      <firstname>Richard</firstname>
      <lastname>Dawkins</lastname>
    </name>
    <address>
      <city>Timbuktu</city>
      <zip>99999</zip>
    </address>
  </author>
</book>
```

**Figure 2**

```xml
<!ELEMENT book (booktitle, author)>
<!ELEMENT article (title, author*, contactauthor)>
<!ELEMENT contactauthor EMPTY>
<!ATTLIST contactauthor authorID ID IDREF IMPLIED>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor (monograph*)>
<!ATTLIST editor name CDATA #REQUIRED>
<!ELEMENT author (name, address)>
<!ATTLIST author id ID #REQUIRED>
<!ELEMENT name (firstname?, lastname?)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT address ANY>
```
XML in RDBMS: DTD Graph

Figure 8
### XML in RDBMS: Schema without inlining

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>booktitle</td>
<td>(booktitleID: integer, booktitle: string)</td>
</tr>
<tr>
<td>article</td>
<td>(articleID: integer, article.contactauthor.authorid: string, article.title: string)</td>
</tr>
<tr>
<td>article.author</td>
<td>(article.authorID: integer, article.author.parentID: integer, article.author.name.firstname: string, article.author.name.lastname: string, article.author.address: string, article.author.authorid: string)</td>
</tr>
<tr>
<td>contactauthor</td>
<td>(contactauthorID: integer, contactauthor.authorid: string)</td>
</tr>
<tr>
<td>title</td>
<td>(titleID: integer, title: string)</td>
</tr>
<tr>
<td>editor</td>
<td>(editorID: integer, editor.parentID: integer, editor.name: string)</td>
</tr>
<tr>
<td>author</td>
<td>(authorID: integer, author.name.firstname: string, author.name.lastname: string, author.address: string, author.authorid: string)</td>
</tr>
<tr>
<td>name</td>
<td>(nameID: integer, name.firstname: string, name.lastname: string)</td>
</tr>
<tr>
<td>firstname</td>
<td>(firstnameID: integer, firstname: string)</td>
</tr>
<tr>
<td>lastname</td>
<td>(lastnameID: integer, lastname: string)</td>
</tr>
<tr>
<td>address</td>
<td>(addressID: integer, address: string)</td>
</tr>
</tbody>
</table>

Figure 10
XML in RDBMS: Schema with aggressive inlining

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>article (articleID: integer, article.contactauthor.isroot: boolean, article.contactauthor.authorid: string)</td>
</tr>
</tbody>
</table>

Figure 11
XML in RDBMS: Query Conversion

- Left: Original Query (in XML-QL and Lorel syntax),
  Right: Converted query

WHERE <book>
    <booktitle> The Selfish Gene </booktitle>
    <author>
        <name>
            <firstname> $f </firstname>
            <lastname> $l </lastname>
        </name>
    </author>
</book> IN * CONFORMING TO pubs.dtd
CONSTRUCT <result> $f $l </result>

Select A."author.name.firstname",
   A."author.name.lastname"
From  author A, book B
Where B.bookID = A.parentID
   AND A.parentCODE = 0
   AND B."book.booktitle" = "The Selfish Gene"

Figure 18
4.2 Converting Simple Recursive Path Expressions to SQL

Consider the following XML-QL query that requires...

```xml
WHERE <*.monograph>
    <editor.(monograph.editor)*>(name> $n </name>
</>
<title> Subclass Cirripedia </title>
IN * CONFORMING TO pubs.dtd
CONSTRUCT <result> $n </result>
```

Select Y.name
From * .monograph X, X.editor.(monograph.editor)* Y
Where X.title = “Subclass Cirripedia”

With Q1 (monographID, name) AS
 (Select X.monographID, X."editor.name"
 From monograph X
 Where X.title = “Subclass Cirripedia”
 UNION ALL
 Select Z.monographID, Z."editor.name"
 From Q1 Y, monograph Z
 Where Y.monographID = Z.parentID AND
     Z.parentCODE = 0
 )
Select A.name
From Q1 A
```

Figure 19
XML in RDBMS: Query Conversion

- Left: Original Query, Right: Converted query

```sql
WHERE <*.monograph>
    <editor.(monograph.editor)*/>
    <name> $n </name>
</>
<title> Subclass Cirripedia </title>
</> IN *
CONSTRUCT <result> $n </result>

Select Y.name
From  *.monograph X, X.editor.(monograph.editor)* Y
Where X.title = "Subclass Cirripedia"

With Q1 (monographID, name) AS
(Select X.monographID, X."editor.name"
From monograph X
Where X.title = "Subclass Cirripedia"
UNION ALL
Select Z.monographID, Z."editor.name"
From Q1 Y, monograph Z
Where Y.monographID = Z.parentID AND
    Z.parentCODE = 0
)
Select A.name
From Q1 A
```

- Note: (right) is a recursive query
  - The query (WITH part) creates a table (Q1) and refers to it in the FROM clause
XML in RDBMS: Converting results to XML

- Some result construction can be done using SQL
- More complex ones require a post-processing step
  - Can be done using a user-defined function or embedded SQL or like
  - Can use "group by" etc to create the appropriate sets
Limitations of RDBMS

- Simple XML queries required too many joins or unions
- No support for sets
  - XML data usually set-valued
- No support for untyped references
  - IDREF is not typed, so storing it is problematic
- No text indices
- Need flexible comparison operators
  - XML treats everything as string
- More powerful recursion
  - SQL3 (latest version) allows recursion
  - Not very commonly used
    - Somewhat hard to reason about