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)
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
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>
      <Neighborhood id="Shadyside">
        ...
      </Neighborhood>
    </City>
  </County>
</State>
```

[[pSpace == parking space ]]
Example XML Fragment for PSF
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 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 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:sequence>
      <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:complexType>
  </xs:element>
</xs:schema>
<?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
    
    `/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
    
    `/State[@id=“Pennsylvania”]/County[@id=“Allegheny”]/Block[count(./pSpace[metered = “yes”]) > 20]`
  
  - Find the cheapest parking spot in Oakland Block 1
    
    `/State[@id=“Pennsylvania”]/County[@id=“Allegheny”]/City[@id=“Pittsburgh”]/Neighborhood[@id=“Oakland”]/Block[@id=‘1’]/pSpace[not(../pSpace/price > ./price)]`
The sample XQuery code below lists the unique speakers in each act of Shakespeare's play Hamlet, encoded in `hamlet.xml`.

```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
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
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)
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*
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
Extensible Markup Language (XML) is a hierarchical data format for information storage. Here is an example of XML text:

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

Figure 1

```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>
```

Figure 2
XML in RDBMS: DTD Graph

Figure 8
The Shared Inlining Technique, hereafter referred to as the Shared Inlining Technique, is a method for improving the performance of database queries. It involves merging related data into a single logical unit, which can then be stored and retrieved more efficiently.

Figure 10

```
booktitle (booktitleID: integer, booktitle: string)
article (articleID: integer, article.contactauthor.authorid: string, article.title: string)
article.author (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)
contactauthor (contactauthorID: integer, contactauthor.authorid: string)
title (titleID: integer, title: string)
editor (editorID: integer, editor.parentID: integer, editor.name: string)
author (authorID: integer, author.name.firstname: string, author.name.lastname: string, author.address: string, author.authorid: string)
name (nameID: integer, name.firstname: string, name.lastname: string)
firstname (firstnameID: integer, firstname: string)
lastname (lastnameID: integer, lastname: string)
address (addressID: integer, address: string)
```
XML in RDBMS: Schema with aggressive inlining

Figure 11

article (articleID: integer, article.contactauthor.isroot: boolean, article.contactauthor.authorid: string)
Figure 18

### Select Y.name.firstname, Y.name.lastname
From book X, X.author Y
Where X.booktitle = “Databases”

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”
4.2 Converting Simple Recursive Path Expressions to SQL

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

```
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
4.2 Converting Simple Recursive Path Expressions to SQL

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

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

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

Figure 19
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