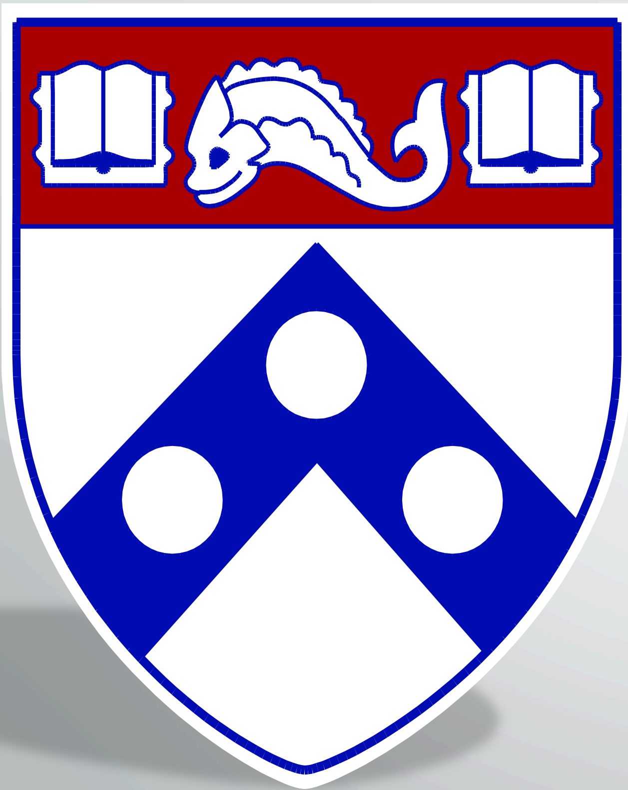


# XML and XQuery

Susan B. Davidson

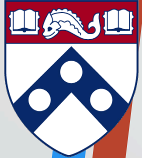
CIS 700: Advanced Topics in Databases

MW 1:30-3

Towne 309

http://www.cis.upenn.edu/~susan/cis700/homepage.html

© 2017 A. Alawini, S. Davidson



# XML Anatomy

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
 <mastersthesis mdate="2002-01-03" key="ms/Brown92">
  <author>Kurt P. Brown</author>
  <title>PRPL: A Database Workload Specification Language</title>
  <year>1992</year>
  <school>Univ. of Wisconsin-Madison</school>
 </mastersthesis>
 <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
  <editor>Paul R. McJones</editor>
  <title>The 1995 SQL Reunion</title>
  <journal>Digital System Research Center Report</journal>
  <volume>SRC1997-018</volume>
  <year>1997</year>
  <ee>db/labs/dec/SRC1997-018.html</ee>
  <ee>http://www.mcjones.org/System_R/SQL_Reunion_95/</ee>
 </article>
```

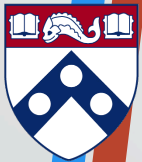
*Processing Instr.*

*Open-tag*

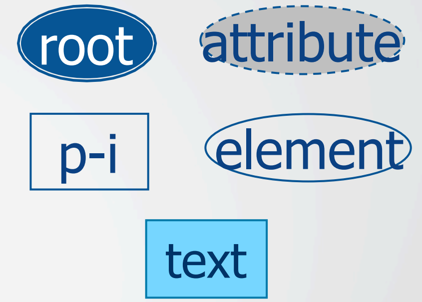
*Element*

*Attribute*

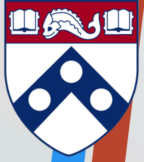
*Close-tag*



# XML Data Model Visualized (and simplified!)




Root
?xml
dblp
mastersthesis
mdate
key
2002…
ms/Brown92
author
title
year
school
Kurt P.…
PRPL…
1992
Univ.…
article
mdate
key
2002…
tr/dec/…
editor
title
journal
volume
year
ee
ee
Paul R.
The…
Digital…
SRC…
1997
db/labs/dec
http://www.



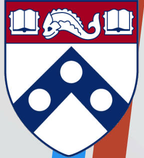
# Structural Constraints: Document Type Definitions (DTDs)

The DTD is an EBNF grammar defining XML structure

- XML document specifies an associated DTD, plus the root element
- DTD specifies children of the root (and so on)

DTD defines special significance for attributes:

- IDs – special attributes that are analogous to keys for elements
- IDREFs – references to IDs
- IDREFS – a nasty hack that represents a list of IDREFs



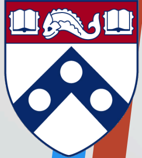
# An Example DTD

Example DTD:

```
<!ELEMENT dblp((mastersthesis | article)*)>
<!ELEMENT mastersthesis(author,title,year,school,committeemember*)>
<!ATTLIST     mastersthesis(mdate   CDATA  #REQUIRED
   key        ID #REQUIRED
   advisor       CDATA  #IMPLIED>
<!ELEMENT author(#PCDATA)>
…
```

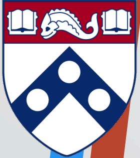
Example use of DTD in XML file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE dblp SYSTEM “my.dtd">
<dblp>…
```



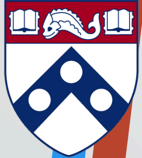
# Representing Graphs and Links in XML: Basically Using Foreign Keys

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE graph SYSTEM “special.dtd">
<graph>
  <author id=“author1”>
    <name>John Smith</name>
  </author>
  <article>
    <author ref=“author1” />  <title>Paper1</title>
  </article>
  <article>
    <author ref=“author1” />  <title>Paper2</title>
  </article>
```



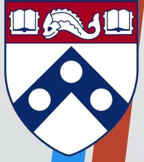
# Graph Data Model

Root
?xml
!DOCTYPE
graph
author
id
author1
name
John Smith
article
author
ref
author1
title
Paper1
article
author
ref
author1
title
Paper2



# Graph Data Model

Root
?xml
!DOCTYPE
graph
author
article
article
id
name
author
title
author
title
author1
John Smith
ref
Paper1
ref
Paper2

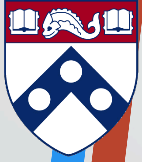


# Querying XML

*How do you query a directed graph? a tree?*

The standard approach used by many XML, semistructured-data, and object query languages:

- Define some sort of a *template* describing traversals from the *root* of the directed graph
- In XML, the basis of this template is called an XPath

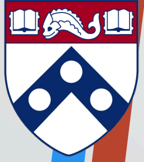


# XPaths

In its simplest form, an XPath is like a path in a file system:

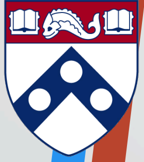
/mypath/subpath/*/morepath

- The XPath returns a *node set* representing the XML nodes (and their subtrees) at the end of the path
- XPaths can have *node tests* at the end, returning only particular node types, e.g., text(), processing-instruction(), comment(), element(), attribute()
- XPath is fundamentally an ordered language: it can query in order-aware fashion, and it returns nodes in order



# Some Example XPath Queries

- /dblp/mastersthesis/title
- /dblp/*/editor
- //title
- //title/text()



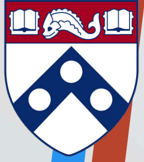
# Context Nodes and Relative Paths

XPath has a notion of a *context* node: it's analogous to a current directory

- "." represents this context node
- ".." represents the parent node
- We can express relative paths:

  subpath/sub-subpath/../.. gets us back to the context node

➢ By default, the document root is the context node



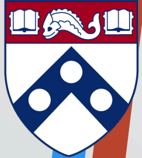
# Predicates – Selection Operations

A *predicate* allows us to filter the node set based on selection-like conditions over sub-XPaths:

/dblp/article[title = “Paper1”]

which is equivalent to:

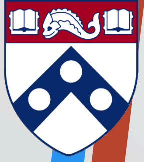
/dblp/article[./title/text() = “Paper1”]



# Axes: More Complex Traversals

Thus far, we've seen XPath expressions that go *down* the tree (and up one step)

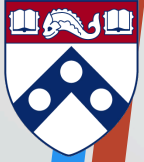
- But we might want to go up, left, right, etc.
- These are expressed with so-called *axes*:
  - self::path-step
  - child::path-step parent::path-step
  - descendant::path-step ancestor::path-step
  - descendant-or-self::path-step ancestor-or-self::path-step
  - preceding-sibling::path-step following-sibling::path-step
  - preceding::path-step following::path-step
- The previous XPaths we saw were in "abbreviated form"



# Querying Order

- We saw in the previous slide that we could query for preceding or following siblings or nodes
- We can also query a node for its position according to some index:
  - fn::first() , fn::last() return index of 0th & last element matching the last step:
  - fn::position() gives the relative count of the current node

```
child::article[fn::position() = fn::last()]
```



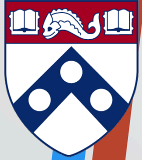
# Beyond XPath: XQuery

A strongly-typed, Turing-complete XML manipulation language

- Attempts to do static typechecking against XML Schema
- Based on an object model derived from Schema

Unlike SQL, fully compositional, highly orthogonal:

- Inputs & outputs collections (sequences or bags) of XML nodes
- Anywhere a particular type of object may be used, may use the results of a query of the same type
- Designed mostly by DB and functional language people



# XQuery's Basic Form

- Has an analogous form to SQL's SELECT..FROM..WHERE..GROUP BY..ORDER BY
- The model: bind nodes (or node sets) to variables; operate over each legal combination of bindings; produce a set of nodes
- "FLWOR" statement [note case sensitivity!]:

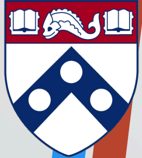
  for {iterators that bind variables}

  let {collections}

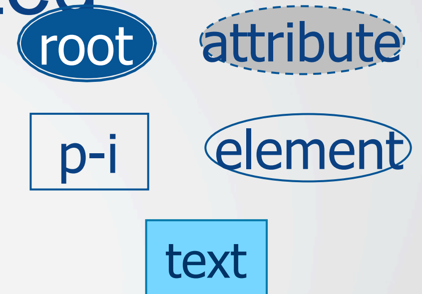
  where {conditions}

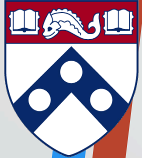
  order by {order-paths}

  return {output constructor}
- Mixes XML + XQuery syntax; use {} as "escapes"



# XML Data Model Visualized



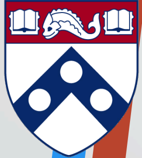


# "Iterations" in XQuery

A series of (possibly nested) FOR statements assigning the results of XPaths to variables

```
for $root in doc ("http://my.org/my.xml")
 for $sub in $root/rootElement,
     $sub2 in $sub/subElement, …
```

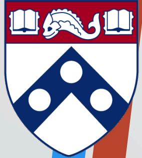
- Something like a template that pattern-matches, produces a "binding tuple"
- For each of these, we evaluate the WHERE and possibly output the RETURN template
- document() or doc() function specifies an input file as a URI



# Two XQuery Examples

```
<root-tag> {
  for $p in doc ("dblp.xml")/dblp/proceedings,
       $yr in $p/yr
  where $yr = "1999"
  return <proc> {$p} </proc>
} </root-tag>
```

```
for $i in doc ("dblp.xml")/dblp/inproceedings[author/text() = "John
  Smith"]
return <smith-paper>
         <title>{ $i/title/text() }</title>
         <key>{ $i/@key }</key>
         { $i/crossref }
   </smith-paper>
```

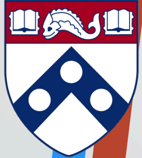


# Nesting in XQuery

Nesting XML trees is perhaps the most common operation

- In XQuery, it's easy – put a subquery in the return clause where you want things to repeat!

```
for $u in doc("dblp.xml")/dblp/university
where $u/country = "USA"
return <ms-theses-99>
        { $u/name } {
         for $mt in $u/../mastersthesis
         where $mt/year/text() = "1999" and ____________
         return $mt/title }
       </ms-theses-99>
```

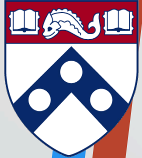


# Collections & Aggregation in XQuery

In XQuery, many operations return collections
- XPaths, sub-XQueries, functions over these, …
- The let clause assigns the results to a variable

Aggregation applies a function over a collection (elegant!)

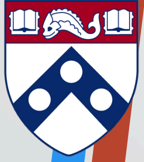
```
let $allpapers := doc (“dblp.xml”)/dblp/article
return <article-authors>
  <count> { fn:count(fn:distinct-values($allpapers/authors)) } </count>
{  for $paper in doc(“dblp.xml”)/dblp/article
   let $pauth := $paper/author
   return <paper> {$paper/title}
          <count> { fn:count($pauth) } </count>
       </paper>
} </article-authors>
```



# Collections, Ctd.

Unlike SQL, we can compose aggregations and create new collections from old:

```
<result> {
let $avgItemsSold := fn:avg(
  for $order in doc("my.xml")/orders/order
  let $totalSold = fn:sum($order/item/quantity)
  return $totalSold)
  return $avgItemsSold
} </result>
```

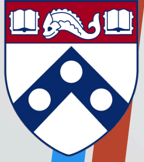


# Distinct-ness

In XQuery, DISTINCT-ness happens as a function over a collection

- But since we have nodes, we can do duplicate removal according to value or node
- Can do fn:distinct-values(collection) to remove duplicate values, or fn:distinct-nodes(collection) to remove duplicate nodes

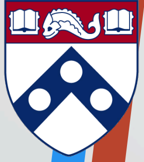
```
for $years in fn:distinct-values(doc(“dblp.xml”)//year/text()
return $years
```



# Sorting in XQuery

- SQL actually allows you to sort its output, with a special ORDER BY clause
- In XQuery, what we order is the sequence of “result tuples” output by the return clause:

```
for $x in doc (“dblp.xml”)/proceedings
order by $x/title/text()
return $x
```



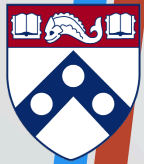
# What If Order Doesn't Matter?

By default:

- SQL is unordered
- XQuery is ordered everywhere!
- But unordered queries are much faster to answer

XQuery has a way of telling the query engine to avoid preserving order:

- ```
  unordered {
    for $x in (mypath) …
  }
  ```
  ```



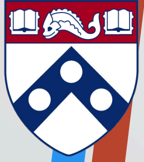
# Querying & Defining Metadata – Can't Do This in SQL

Can get a node's name by querying name():

```
for $x in doc ("dblp.xml")/dblp/*
return name($x)
```

Can construct elements and attributes using computed names:

```
for $x in doc ("dblp.xml")/dblp/*,
 $year in $x/year,
 $title in $x/title/text()
return
element { name($x) } {
 attribute { "year-" + $year } { $title }
}
```



# XQuery Summary

Very flexible and powerful language for XML

- Clean and orthogonal: can always replace a collection with an expression that creates collections
- DB and document-oriented (with keyword search extensions)
- The core is relatively clean and easy to understand

Turing Complete – there are several XQuery functions that enable this (not discussed).