Gaining Access to Diverse Data

We have focused on data integration in the relational model

Simplest model to understand

Real-world data is often not in relational form

e.g., Excel spreadsheets, Web tables, Java objects, RDF, ...

- One approach: convert using custom wrappers (Ch. 9)

- But suppose tools would adopt a standard export (and import) mechanism?

  ... This is the role of XML, the eXtensible Markup Language
What Is XML?

Hierarchical, human-readable format
- A “sibling” to HTML, always parsable
- “Lingua franca” of data: encodes documents and structured data
- Blends data and schema (structure)

Core of a broader ecosystem
- Data – XML (also RDF, Ch. 12)
- Schema – DTD and XML Schema
- Programmatic access – DOM and SAX
- Query – XPath, XSLT, XQuery
- Distributed programs – Web services

Diagram:
- XQuery
- XPath
- SAX/DOM
- HTTP
- DTD/Schema
- Database
- Document
- Web Service
- REST / SOAP + WSDL
- Procedural language (Java, JavaScript, C++, …)
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>
</dblp>
XML Data Components

XML includes two kinds of data items:

Elements

- Hierarchical structure with open tag-close tag pairs
- May include nested elements
- May include attributes within the element’s open-tag
- Multiple elements may have same name
- Order matters

```
<article mdate="2002-01-03" ...
  <editor>Paul R. McJones</editor>...
</article>
```

Attributes

- Named values – not hierarchical
- Only one attribute with a given name per element
- Order does NOT matter

```
mdate="2002-01-03"
```
Well-Formed XML: Always Parsable

Any legal XML document is always parsable by an XML parser, without knowledge of tag meaning

- The start – *preamble* – tells XML about the char. encoding
  
  ```xml
  <?xml version="1.0" encoding="utf-8"?>
  ```

- There’s a single root element

- All open-tags have matching close-tags (unlike many HTML documents!), or a special:
  
  ```xml
  <tag/> shortcut for empty tags (equivalent to <tag></tag>)
  ```

- Attributes only appear once in an element

- XML is case-sensitive
Outline

- XML data model
  - Node types
  - Encoding relations and semi-structured data
  - Namespaces
- XML schema languages
- XML querying
- XML query processing
- XML schema mapping
XML as a Data Model

XML “information set” includes 7 types of nodes:

- Document (root)
- Element
- Attribute
- Processing instruction
- Text (content)
- Namespace
- Comment

XML data model includes this, plus typing info, plus order info and a few other things
XML Data Model Visualized (and simplified!)

Root
  - ?xml
  - dblp

mastersthesis
  - mdate
  - key
  - author
  - title
  - year
  - school
  - mdate
  - key

article
  - editor
  - title
  - journal
  - volume
  - year
  - ee

2002...
ms/Brown92
PRPL...
Kurt P....

1992
Univ....

2002...
tr/dec/...
The...
Paul R.

1997
db/labs/dec
http://www.
XML Easily Encodes Relations

Student-course-grade

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>exp-grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>570</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>550</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>103</td>
<td></td>
</tr>
</tbody>
</table>

```
@student-course-grade
	<tuple sid="1" cid="570103" exp-grade="B"/>
	<tuple sid="23" cid="550103" exp-grade="A"/>
</student-course-grade>
```
XML is “Semi-Structured”

<p>...<br/>

```xml
<parents>
  <parent name="Jean">
    <son>John</son>
    <daughter>Joan</daughter>
    <daughter>Jill</daughter>
  </parent>
  <parent name="Feng">
    <daughter>Ella</daughter>
  </parent>
...```
Combining XML from Multiple Sources with the Same Tags: Namespaces

- **Namespaces** allow us to specify a context for different tags
- Two parts:
  - Binding of namespace to URI
  - Qualified names

```xml
<root xmlns="http://www.first.com/aspace" xmlns:otherns="...">
  <myns:tag xmlns:myns="http://www.fictitious.com/mypath">
    <thistag>is in the default namespace (www.first.com/aspace)</thistag>
    <myns:thistag>is in myns</myns:thistag>
    <otherns:thistag>is a different tag in otherns</otherns:thistag>
  </myns:tag>
</root>
```

**Default namespace for non-qualified names**

**Defines “otherns” qualifier**
Outline

- XML data model
- XML schema languages
  - DTDs
  - XML Schema (XSD)
- XML querying
- XML query processing
- XML schema mapping
XML Isn’t Enough on Its Own

It’s too unconstrained for many cases!
- How will we know when we’re getting garbage?
- How will we know what to query for?
- How will we understand what we received?

We also need:
- An idea of (at least part of) the structure
- Some knowledge of how to interpret the tags...
Structural Constraints: Document Type Definitions (DTDs)

The DTD is an EBNF grammar defining XML structure

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

DTD also defines special attribute types:

- **IDs** – special attributes that are analogous to keys for elements
- **IDREFs** – references to IDs
- **IDREFS** – a list of **IDREFs**, space-delimited (!)
- All other attributes are essentially treated as **strings**
An Example DTD and How to Reference It from XML

Example DTD:

```xml
<!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
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE dblp SYSTEM "my.dtd">
<dblp>...
```
Links in XML: Restricted Foreign Keys

```xml
<?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>
```

Suppose we have defined this to be of type ID

Suppose we have defined this to be of type IDREF
The Limitations of DTDs

DTDs capture grammatical structure, but have some drawbacks:

- Don’t capture database datatypes’ domains
- IDs aren’t a good implementation of keys
  - Why not?
- No way of defining OO-like inheritance

- “Almost XML” syntax – inconvenient to build tools for them
XML Schema (XSD)

Aims to address the shortcomings of DTDs

- XML syntax
- Can define keys using XPaths (we’ll discuss later)
- Type subclassing that also includes restrictions on ranges
  - “By extension” (adds new data) and “by restriction” (adds constraints)
- ... And, of course, domains and built-in datatyps

(Note there are other XML schema formats like RELAX NG)
Basics of XML Schema

Need to use the XML Schema namespace (generally named xsd)

- **simpleTypes** are a way of restricting domains on scalars
  - Can define a `simpleType` based on integer, with values within a particular range

- **complexType**es are a way of defining element/attribute structures
  - Basically equivalent to `!ELEMENT`, but more powerful
  - Specify sequence, choice between child elements
  - Specify `minOccurs` and `maxOccurs` (default 1)

- Must associate an element/attribute with a `simpleType`, or an element with a `complexType`
Simple XML Schema Example

Associates “xsd” namespace with XML Schema

This is the root element, with type specified below

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="mastersthesis" type="ThesisType"/>

  <xsd:complexType name="ThesisType">
    <xsd:attribute name="mdate" type="xsd:date"/>
    <xsd:attribute name="key" type="xsd:string"/>
    <xsd:attribute name="advisor" type="xsd:string"/>
    <xsd:sequence>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="year" type="xsd:integer"/>
      <xsd:element name="school" type="xsd:string"/>
      <xsd:element name="committeemember" type="CommitteeType"
        minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
Designing an XML Schema/DTD

Not as formalized as relational data design

- Typically based on an existing underlying design, e.g., relational DBMS or spreadsheet

We generally orient the XML tree around the “central” objects

Big decision: element vs. attribute

- Element if it has its own properties, or if you might have more than one of them
- Attribute if it is a single property – though element is OK here too!
Outline

✓ XML data model
✓ XML schema languages
➢ XML querying
  ▪ DOM and SAX
  ▪ XPath
  ▪ XQuery
▪ XML query processing
▪ XML schema schema mapping
A huge benefit of XML – standard parsers and standard (cross-language) APIs for processing it

DOM: an object-oriented representation of the XML parse tree (roughly like the Data Model graph)

- **DOM objects** have methods like “getFirstChild()”, “getNextSibling”
- Common way of traversing the tree
- Can also modify the DOM tree – alter the XML – via insertAfter(), etc.

Sometimes we don’t want all of the data: SAX

- **Parser interface** that calls a function each time it parses a processing-instruction, element, etc.
- Your code can determine what to do, e.g., build a data structure, or discard a particular portion of the data
Alternate approach to processing the data: a *query* language

- Define some sort of a *template* describing traversals from the *root* of the directed graph

- Potential benefits in parallelism, views, schema mappings, and so on

- In XML, the basis of this template is called an XPath
  - Can also declare some constraints on the values you want
  - The XPath returns a *node set* of matches
In its simplest form, an Xpath looks like a path in a file system:

[/mypath/subpath/*/morepath]

- But 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, filtering all except node types
  - `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
Recall Our Sample XML

```xml
<?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>
</dblp>
```
Recall Our XML Tree
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:
  \texttt{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. via *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”
  
  /child::dblp/child::mastersthesis/child::title
  /descendant-or-self::title
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’s position according to some index:
  - `fn::first()`, `fn::last()` index of 0th & last element matching the last step
  - `fn::position()` relative count of the current node

```
child::article[fn::position() = fn::last()]
```
XPath Is Used within Many Standards

- XML Schema uses simple XPaths in defining keys and uniqueness constraints
- XQuery
- XSLT
- XLink and Xpointer – hyperlinks for XML
XPath is used to express XML schema keys & foreign keys.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="ThesisType">
    <xsd:attribute name="key" type="xsd:string"/>
    <xsd:sequence>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="school" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="dblp">
    <xsd:sequence>
      <xsd:element name="mastersthesis" type="ThesisType">
        <xsd:keyref name="schoolRef" refer="schoolId">
          <xsd:selector xpath="./school"/>
          <xsd:field xpath="./text()"/>
        </xsd:keyref>
      </xsd:element>
      <xsd:element name="university" type="SchoolType"/>
    </xsd:sequence>
    <xsd:key name="schoolId">
      <xsd:selector xpath="university"/> <xsd:field xpath="@key"/>
    </xsd:key>
  </xsd:element>
</xsd:schema>
```

Foreign key refers to key by its ID.

Item w/key = selector

Field is its key.
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

Can be used to define queries, views, and (using a subset) schema mappings
XQuery’s Basic Form

- Has an analogous form to SQL’s
  \texttt{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!]:
  \begin{verbatim}
  for \{iterators that bind variables\}
  let \{collections\}
  where \{conditions\}
  order by \{order-paths\}
  return \{output constructor\}
\end{verbatim}
- Mixes XML + XQuery syntax; use \{\} as “escapes”
Recall Our XML Tree

```
<root>
  <xml>
    <dblp>
      <mastersthesis>
        <mdate>2002</mdate>
        <key>ms/Brown92</key>
        <author>Kurt P.</author>
        <title>PRPL…</title>
        <school>Univ….</school>
      </mastersthesis>
      <article>
        <mdate>2002</mdate>
        <key>tr/dec/…</key>
        <title>The…</title>
        <journal>DBPL…</journal>
        <year>1997</year>
        <ee>db/labs/dec</ee>
      </article>
    </dblp>
  </xml>
</root>
```
“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
  - Early versions used “document”; modern versions use “doc”
Two XQuery Examples

```
<root-tag>{
    for $p in doc ("dblp.xml")/dblp/article,
        $yr in $p/yr
    where $yr = "1997"
    return <paper> { $p/title } </paper>
} </root-tag>

for $i in doc ("dblp.xml")/dblp/article[author/text() = "John Smith"]
return <smith-paper>
    <title>{ $i/title/text() }</title>
    <key>{ $i/@key } </key>
    { $i/crossref }
</smith-paper>
```
Restructuring Data 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 doc("dblp.xml")/dblp/mastersthesis
    where $mt/year/text() = "1999" and $mt/school = $u/name
    return $mt/title }
</ms-theses-99>
```
In XQuery, many operations return **collections**

- XPaths, sub-XQueries, functions over these, ...
- The **let** clause assigns the results to a variable

Aggregation simply applies a function over a collection, where the function returns a value (very elegant!)

```xquery
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>
```
Unlike in SQL, we can compose aggregations and create new collections from old:

```xml
<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

```xquery
for $years in fn:distinct-values(doc("dblp.xml")//year/text())
return $years
```
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
```
Querying & Defining Metadata

Can get a node’s name by querying `name()`:

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

Can construct elements and attributes using **computed names**:

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

- A view is a named query
- We use the name of the view to invoke the query (treating it as if it were the relation it returns)

XQuery:
```xquery
declare function V() as element(content)* {
    for $r in doc("R")/root/tree,
        $a in $r/a, $b in $r/b, $c in $r/c
    where $a = "123"
    return <content>{$a, $b, $c}</content>
}
```

Using the view:
```xquery
for $v in V()//content,
for $r in doc("r")/root/tree
    where $v/b = $r/b
return $v
```
Outline

- ✔ XML data model
- ✔ XML schema languages
- ✔ XML querying
- ➢ XML query processing
- ▪ XML schema mapping
Streaming Query Evaluation

- In data integration scenarios, the query processor must fetch **remote** data, parse the XML, and process.

- Ideally: we can **pipeline** processing of the data as it is “streaming” to the system.

  “Streaming XPath evaluation”

  ... which is also a building block to pipelined XQuery evaluation...
Main Observations

- XML is sent (serialized) in a form that corresponds to a left-to-right depth-first traversal of the parse tree.

- The “core” part of XPath (child, descendant axes) essentially corresponds to regular expressions over edge labels.
The First Enabler: SAX (Simple API for XML)

- If we are to match XPaths in streaming fashion, we need a stream of XML nodes

- SAX provides a series of event notifications
  - Events include open-tag, close-tag, character data

  - Events will be fired in depth-first, left-to-right traversal order of the XML tree
The Second Key: Finite Automata

- Convert each XPath to an equivalent regular expression
- Build a finite automaton (NFA or DFA) for the regexp

```
/dblp/article

//year
```
Matching an XPath

- Assume a “cursor” on active state in the automaton
- On matching open-tag: push advance active state
- On close-tag: pop active state

<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
   <mastersthesis>
       ...
   </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>
</dblp>
Matching an XPath

- Assume a “cursor” on active state in the automaton
- On matching open-tag: push advance active state
- On close-tag: pop active state

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
  <mastersthesis>
    ...
  </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>
</dblp>
```

event: start-element “mastersthesis”
Matching an XPath

- Assume a "cursor" on active state in the automaton
- On matching open-tag: push advance active state
- On close-tag: pop active state

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
  <mastersthesis>...
  </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>
</dblp>
```
Matching an XPath

- Assume a “cursor” on active state in the automaton
- On matching open-tag: push advance active state
- On close-tag: pop active state

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
  <mastersthesis>
    ...
  </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>
</dblp>
```

Event: start-element "article"

match!
Different Options

- Many different “streaming XPath” algorithms
  - What kind of automaton to use
    - DFA, NFA, lazy DFA, PDA, proprietary format
  - Expressiveness of the path language
    - Full regular path expressions, XPath, ...
    - Axes

- Which operations can be pushed into the operator
  - XPath predicates, joins, position predicates, etc.
From XPaths to XQueries

- An XQuery takes multiple XPaths in the FOR/LET clauses, and iterates over the elements of each XPath (binding the variable to each)
  
  ```
  FOR $rootElement in doc("dblp.xml")/dblp,
      $rootChild in $rootElement/article[author="Bob"],
      $textContent in $rootChild/text()
  ```

- We can think of an XQuery as doing **tree matching**, which returns tuples ($i, $j) for each tree matching $i and $j in a document.

- Streaming XML path evaluator that supports a hierarchy of matches over an XML document.
**XQuery Path Evaluation**

- Multiple, dependent state machines outputting *binding tuples*

FOR $rootElement in doc("dblp.xml")/dblp,
    $rootChild in $rootElement/article[author="Bob"],
    $textContent in $rootChild/text()

Evaluate a pushed-down selection predicate
Beyond the Initial FOR Paths

- The streaming XML evaluator operator returns tuples of bindings to nodes

<table>
<thead>
<tr>
<th>$rootElement</th>
<th>$rootChild</th>
<th>$textContent</th>
</tr>
</thead>
</table>

- We can now use standard relational operators to join, sort, group, etc.

- Also in some cases we may want to do further XPath evaluation against one of the XML trees bound to a variable
Creating XML

- To return XML, we need to be able to take streams of binding tuples and:
  - Add tags around certain columns
  - Group tuples together and nest them under tags

- Thus XQuery evaluators have new operators for performing these operations
An Example XQuery Plan

XML output operator

XPath evaluation against a binding

Relational-style query operators (outerjoin)

Streaming XPath evaluation

XPath matcher

XML tagging

$rootElement

$rootChild

set

text()

$textContent

Σ

"Bob"

XML tagging

XPath matcher

$editor

$title

XML grouping

XPath matcher

$rootChild

$dblp.xml

<dblp>...

<editor>Paul R. McJones</editor>

<title>The 1995/elipsis</title>

$textContent

Π

(<article>/elipsis</article>,

["Paul R. McJones","The 1995/elipsis", /elipsis])

Π

(<BobResult><editor>Paul R. McJones</editor>


Relational-style query operators (outerjoin)

XPath evaluation against a binding

XML output operator
Optimizing XQueries

- An entire field in and of itself

- A major challenge versus relational query optimization: estimating the “fan-out” of path evaluation

- A second major challenge: full XQuery supports arbitrary recursion and is Turing-complete
Outline

- XML data model
- XML schema languages
- XML querying
- XML query processing
- XML schema mapping
In Chapter 3 we saw how schema mappings were described for relational data:

- As a set of constraints between source and target databases.

In the XML realm, we want a similar constraint language, but must address:

- Nesting – XML is hierarchical
- Identity – how do we merge multiple partial results into a single XML tree?
One Approach: Piazza XML Mappings

Derived from a subset of XQuery extended with node identity
  - The latter is used to merge results with the same node ID
Directional mapping language based on annotations to XML templates

An output element in the template, ~ XQuery RETURN

<output>
{:: $var IN document("doc")/path WHERE condition :}
<tag>$var</tag>
</output>

Create the element for each match to this set of XPaths & conditions
Populate with the value of a binding

- Translates between parts of data instances
- Supports special annotations and object fusion
Mapping Example between Two XML Schemas

Target: *Publications by book*

```xml
<pubs>
  <book>*
    <title>
    <author>*
      <name>
```

Source: *Publications by author*

```xml
<authors>
  <author>*
    <full-name>
    <publication>*
      <title>
      <pub-type>
```

Has an entity-relationship model representation like:

- **publication**
  - **title**
  - **pub-type**

- **author**
  - **name**

writtenBy
Example Piazza-XML Mapping

<pre>&lt;pubs&gt;
  &lt;book&gt;
    {: $a IN document("...")/authors/author,
       $an IN $a/full-name,
       $t IN $a/publication/title,
       $typ IN $a/publication/pub-type
       WHERE $typ = "book" :}

    &lt;title&gt;{$t}&lt;/title&gt;
    &lt;author&gt;&lt;name&gt;{$an}&lt;/name&gt;&lt;/author&gt;
  &lt;/book&gt;
&lt;/pubs&gt;
</pre>

Output one book per match to author

Insert title and author name subelements
Example Piazza-XML Mapping

```xml
<pubs>
  <book piazza:id={$t}>
    {: $a IN document("...")/authors/author,
      $an IN $a/full-name,
      $t IN $a/publication/title,
      $typ IN $a/publication/pub-type
      WHERE $typ = "book" :}

  <title piazza:id={$t}>{$t}</title>
  <author><name>{$an}</name></author>
  </book>
</pubs>
```

- **Output one book per match to author**
- **Insert title and author name subelements**
- **Merge elements if they are for the same value of $t**
The underpinnings of the Piazza-XML mapping language can be captured using *nested tuple-generating dependencies* (nested TGDs)

- Recall relational TGDs from Chapter 3

\[
\forall X, Y, S(\phi(X, Y) \land \Phi(S) \rightarrow \exists Z, T(\psi(X, Z) \land \Psi(T)))
\]

- As before, we’ll typically omit the \( \forall \) quantifiers...
Example Piazza-XML Mapping as a Nested TGD

\[
\text{authors}(\text{author}) \land \text{author}(f, \text{publication}) \land \text{publication}(t, \text{book}) \rightarrow \\
\exists p (\text{pubs}(\text{book}) \land \text{book}(t, \text{author}', \text{publisher}) \land \text{author}'_{t,f}(f) \land \text{publisher}_{t}(p))
\]

Grouping keys in target
Query Reformulation for XML

- Two main versions:
  - Global-as-view-style:
    - Query is posed over the target of a nested TGD, or a Piazza-XML mapping
    - Can answer the query through standard XQuery view unfolding
  - Bidirectional mappings, more like GLAV mappings in the relational world:
    - An advanced topic – see the bibliographic notes
XML Wrap-up

- XML forms an important part of the data integration picture – it’s a “bridge” enabling rapid connection to external sources

- It introduces new complexities in:
  - Query processing – need streaming XPath / XQuery evaluation
  - Mapping languages – must support identity and nesting
  - Query reformulation

- It also is a bridge to RDF and the Semantic Web (Chapter 12)