Introduction to XSLT
Processing the XML data

• Huge amount of XML information, and growing
• We need to “manage” it, and then “process” it
  • *Store it efficiently*
  • Verify the correctness
  • *Filter, search, select, join, aggregate*
  • *Create new pieces of information*
  • Clean, normalize the data
  • *Update it*
  • Take actions based on the existing data
  • *Write complex execution flows*

• No conceptual organization like for relational databases (applications are too heterogeneous)
Frequent solutions to XML data management

- Map it to *generic* programming APIs (e.g. DOM, SAX, StaX)
- *Manually* map it to *non-generic* APIs
- *Automatically* map it to *non-generic* structures
- Use *XML extensions* of existing languages
- Shredding for relational stores
- *Native XML processing through XSLT and XQuery*
History of XSLT

• Much older then XQuery
  – XSLT 1.0 and XSLT 2.0

• Was designed a re-formatting language for the browsers
  – Still primarily used in this way (e.g. Ebay has more 10,000 XSLT stylesheets)
  – Most browsers have an embedded XSLT processor
  – Now has broader applications for XML management

• XSLT 2.0 and XQuery 1.0 are designed jointly

• Same data model, same type system, same Xpath 2.0

• Different programming paradigm
  – XQuery is compositional and functional, XSLT is based on recursive templates
XQuery, XPath, XSLT

- **XPath 1.0**
  - FLWOR expressions
  - Node constructors
  - Validation
  - Year: 1999

- **XPath 2.0**
  - Extends XPath 1.0
  - Year: 2007

- **XQuery 1.0**
  - Extends XPath 2.0, almost backwards compatible

- **XSLT 1.0**
  - Uses XPath 1.0

- **XSLT 2.0**
  - Uses XQuery 1.0

03/09/07
Your first XSLT stylesheet

<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/TR/xhtml1/strict">
  <xsl:template match="/">
    <html>
      <head>
        <title>Expense Report Summary</title>
      </head>
      <body>
        <p>Total Amount: <xsl:value-of select="expense-report/total"/>
        </p>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
The anatomy of a stylesheet

• An XSLT program is an XML document
• The root element of the document is called \texttt{xsl:stylesheet} and is composed of a set of “\texttt{templates}” (i.e. elements called \texttt{xsl:template})
• The \texttt{xsl} namespace is bound to the “official” XSLT URI (e.g. \url{http://www.w3.org/1999/XSL/Transform})
• The XML elements composing the XSLT program: a blend of “\texttt{user}” names and “\texttt{XSLT}” names (QNames in the \texttt{xsl} namespace)
• The “simple” \texttt{xsl} elements are “interpreted” and replaced by the result of their evaluation
  - \texttt{xsl:for-each}, \texttt{xsl:if}, \texttt{xsl:choose}, \texttt{xsl:value-of}
Each template rule specifies how certain nodes from the input XML doc have to be reformatted in the output.
XSLT templates

• Rules that describe how certain input XML nodes have to be transformed in the output nodes
• Represented by elements in the xsl namespace called <xsl:template>
• Can have patterns that describe to what kind of nodes is the template rule applicable
  – <xsl:template match="author | editor">
• Can have names (later)
• Have a body -- an XML fragment that described the output
  – <xsl:template match="*"> <foobars/> <xsl:template>
Template patterns

• Describe to what kind of nodes is a template applicable to

• Represented as an optional `match` attribute on the `xsl:template` elements

• The value of the match attribute is a string representing a pattern

• The pattern language is a subset of the XPath language

• A node `matches` a pattern if it is a member of the result list of nodes of the pattern expression (almost normal XQuery evaluation)
Template patterns: examples

- `para` matches any `para` element
- `*` matches any element
- `chapter|appendix` matches any `chapter` element and any `appendix` element
- `olist/item` matches any `item` element with an `olist` parent
- `appendix//para` matches any `para` element with an `appendix` ancestor element
- `/` matches the root node
- `text()` matches any text node
- `processing-instruction()` matches any processing instruction
- `node()` matches any node other than an attribute node and the root node
- `id("W11")` matches the element with unique ID `W11`
- `para[1]` matches any `para` element that is the first `para` child element of its parent
- `@class` matches any `class` attribute (not any element that has a `class` attribute)
- `@*` matches any attribute
Applying a template to a single node

• Input
  – <foo>beam</foo>

• Two example templates
  <xsl:template match="foo">                                <bar>baz</bar>
    <bar>baz</bar>
  </xsl:template>
  <xsl:template match="foo">
    <bar><xsl:value-of select="."</xsl:value-of></bar>            <bar>beam</bar>
  </xsl:template>

• Applying a template to a single node
  – Return the body of the template
  – All the xsl elements in the body are “interpreted” and replaced by their result
  – The other elements remain unchanged
  – The current node is set to the input node while evaluating Xpath expressions (remember “.”?)
Recursive application of templates

• The templates are not (normally) invoked by hand (see later)
• XSLT semantics is based on a built-in, recursive application of templates
  • \texttt{Apply-templates( list of XML nodes) -> list of XML nodes}
    – For each input node (in order, see later)
      • Find the “best” template that applies (see conflicting templates later…)
      • Note: choice of template is on a node basis
      • Apply the template, returns back a sequence of nodes
    – Concatenate all partial results, return
• The evaluation of the XSLT main program starts by invoking this recursive procedure on the input document node
Invoking the recursive application of templates

• Why is this procedure “recursive”?  
  – While evaluating a template one can trigger the re-evaluation of this procedure
  – `<xsl:apply-template>`

• Example:
  – Input
    • This is an `<emph>important</emph>` point.
  – Template
    ```xml
    <xsl:template match="emph">
        <fo:inline-sequence font-weight="bold">
            <xsl:apply-templates select="./text()"/>
        </fo:inline-sequence>
    </xsl:template>
    ```
xsl:apply-templates

• Re-enter the built-in recursive application of templates

• Has a *select* attribute that specifies on what set of nodes to apply the procedure (using Xpath)
  – <xsl:apply-templates select="author"/>
  – <xsl:apply-templates select="author/name"/>
  – <xsl:apply-templates select=".//heading"/>
  – <xsl:apply-templates select="ancestors::*:department/group"/>
  – <xsl:apply-templates select="."/>

• The order of those nodes can be changed using a xsl:sort (see later); default is document order

• If no *select* attribute, then implicitly trigger the recursive application of templates on the list of children of the current node
Default templates

• What happens if there is no template that matches a node? Default templates..

• Elements and document nodes

```
<xsl:template match="*|/">  <xsl:apply-templates/>
</xsl:template>
```

• Attributes and text nodes

```
<xsl:template match="text()|@*">  <xsl:value-of select="."/>
</xsl:template>
```

• The other nodes

```
<xsl:template match="processing-instruction()|comment()"/>
```
Named templates

• Sometimes one can invoke a particular template -- by name
  <xsl:template name="authorsTemplate">
  • Instead of <xsl:apply-templates>
    – <xsl:call-template name="authorsTemplate">
  • Semantics is the same
  • Small semantic difference
    – xsl:call-templates does not change the current node
    – xsl:apply-templates does
xsl:value-of

• You have seen it already
• `<xsl:value-of select="path expression"/>`
• Evaluates the path expression => nodes
• Apply the fn:string(..) function to each node (remember it ?)
• Concatenate the strings
• Create (and return) a new text node with this value
xsl:for-each

<xsl:for-each
  select = node-set-expression>
  <!-- Content: (xsl:sort*, template-body) -->
</xsl:for-each>

- The node-set expression evaluates to a list of nodes
- For each one of them return the template body, evaluated normally
- Each application returns a list of nodes, concatenate them all
- The input list is processed in document order in case of no sort, otherwise in the sorting specified by the xsl:for-each
Xsl:for-each example

**Data**

```xml
<customers>
  <customer>
    <name>...</name>
    <order>...</order>
    <order>...</order>
  </customer>
  <customer>
    <name>...</name>
    <order>...</order>
    <order>...</order>
  </customer>
</customers>
```

**Program**

```xml
<xsl:template match="/">
  <html>
    <head>
      <title>Customers</title>
    </head>
    <body>
      <table>
        <tbody>
          <xsl:for-each select="customers/customer">
            <tr>
              <th>
                <xsl:apply-templates select="name"/>
              </th>
              <xsl:for-each select="order">
                <td>
                  <xsl:apply-templates/>
                </td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </tbody>
      </table>
    </body>
  </html>
</xsl:template>
```
Xsl:if

• General form

```xml
<xsl:if
test = boolean-expression>
<!-- Content: template-body -->
</xsl:if>
```

• Example 1:

```xml
<xsl:template match="namelist/name">
  <xsl:apply-templates/>
  <xsl:if test="not(position()=last())">, </xsl:if>
</xsl:template>
```

• Example 2:

```xml
<xsl:template match="item">
  <tr>
    <xsl:if test="position() mod 2 = 0">
      <xsl:attribute name="bgcolor">yellow</xsl:attribute>
    </xsl:if>
    <xsl:apply-templates/>
  </tr>
</xsl:template>
```
Xsl:choose

• General form:

```xml
<xsl:choose>
  <!-- Content: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>

<xsl:when
  test = boolean-expression>
  <!-- Content: template-body -->
</xsl:when>

<xsl:otherwise>
  <!-- Content: template-body -->
</xsl:otherwise>
```
xsl:choose: example

<xsl:template match="orderedlist/listitem">
  <fo:list-item indent-start='2pi'>
    <fo:list-item-label>
      <xsl:variable name="level" select="count(ancestor::orderedlist) mod 3"/>
      <xsl:choose>
        <xsl:when test='$level=1'>
          <xsl:number format="i"/>
        </xsl:when>
        <xsl:when test='$level=2'>
          <xsl:number format="a"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:number format="1"/>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:text>. </xsl:text>
    </fo:list-item-label>
    <fo:list-item-body>
      <xsl:apply-templates/>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>
xsl:sort

• Can be a child of xsl:apply-templates or xsl:for-each elements; multiple xsl:sort possible

• General form:
  <xsl:sort
    select = string-expression
    lang = { nmtoken }
    data-type = { "text" | "number" | qname-but-not-ncname }
    order = { "ascending" | "descending" }
    case-order = { "upper-first" | "lower-first" } />

• Example:
  <xsl:template match="employees">
    <ul>
      <xsl:apply-templates select="employee">
        <xsl:sort select="name/family"/>
        <xsl:sort select="name/given"/>
      </xsl:apply-templates>
    </ul>
  </xsl:template>

  <xsl:template match="employee">
    <li>
      <xsl:value-of select="name/given"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="name/family"/>
    </li>
  </xsl:template>
Xsl:key

- Xsl:key allows to declare “keys” (unique identifiers for elements belonging to a certain domain)
- A key is composed of:
  - the name of the key (an expanded-name)
  - the set of nodes which has the key (the domain)
  - the value of the key (a string)
- General form:
  ```xml
  <xsl:key
    name = qname
    match = pattern
    use = expression />
  ```
- Can appear at the root of a stylesheet
- Use later using the `key(Qname, expression)` function
- Can be used to create cross-references between elements (e.g. bibliographical references, HTML links, etc)
xsl:key: example

• One bib.xml document containing bibliographical entries:
  
  `<entry name="Java">...</entry>`

• One document containing references:
  
  `<bibref>Java</bibref>`

• The stylesheet formatting the bibrefs

  `<xsl:template match="bibref">
    <xsl:variable name="name" select="."/>
    <xsl:for-each select="document('bib.xml')">
      <xsl:apply-templates select="key('bib',$name)"/>
    </xsl:for-each>
  </xsl:template>`
Creating output: elements

- Constant elements (similar to the direct element constructors in XQuery)
  
  ```xml
  <xsl:if test="a mod 2 = 0">
    <foobar>baz</foobar>
  </xsl:if>
  ```

- Computed element generation (similar to the computed element constructors in XQuery)

- General form:
  
  ```xml
  <xsl:element
    name = { qname }
    namespace = { uri-reference }
    use-attribute-sets = qnames>
    <!-- Content: template-body -->
  </xsl:element>
  ```

- Example:
  
  - `<xsl:element name="foobar">baz</xsl:element>`
  - `<xsl:element name="foobar"><xsl:value-of select="./text()"/></xsl:element>`
Creating output: attributes

• Computed attribute generation (similar to the computed attribute constructors in XQuery)

• General form:
  
```xml
<xsl:attribute
  name = { qname }
  namespace = { uri-reference }
  <!-- Content: template-body -->
</xsl:attribute>
```

• Example:
  
```xml
  – <xsl:attribute name="foobar">baz</xsl:element>
  – <xsl:attribute name="foobar"><xsl:value-of select="./text()"/></xsl:attribute>
```
Creating output: text

- `<xsl:value-of select="expression">`
- `<xsl:text> XXXX </xsl:text>`
Attribute value templates

• Similar to the dynamically computed content in the direct attribute constructor in XQuery
  
  <foo bar="baz{$x}"/>

• Unlike Xquery, only works for attributes (xsl:value-of does the job for the elements)

• Use “{“ and “}” like Xquery (or vice-versa..)

• Example

  <xsl:variable name="image-dir">/images</xsl:variable>

  <xsl:template match="photograph">
    <img src="{$image-dir}/{href}" width="{size/@width}"/>
  </xsl:template>

  <photograph>
    <href>headquarters.jpg</href>
    <size width="300"/>
  </photograph>

  <img src="/images/headquarters.jpg" width="300"/>
xsl:copy and xsl:copy-of

• **xsl:copy**
  - Creates a *shallow* copy of the current node -- copying the shell (name + NS) but not attributes and children
  ```xml
  <xsl:copy
    use-attribute-sets = qnames>
    <!-- Content: template-body -->
  </xsl:copy>
  ```

• **Example -- the identity transformation**
  ```xml
  <xsl:template match="@*|node()">  
    <xsl:copy>
      <xsl:apply-templates select="@*|node()/">
      </xsl:copy>
  </xsl:template>
  ```

• **xsl:copy-of --- a way of copying *entire* XML tree fragments**
Variables and parameters

- **General form:**
  ```xml
  <xsl:variable
    name = qname
    select = expression>
    <!-- Content: template-body-->
  </xsl:variable>
  ```

- Similar to the LET in Xquery
- Variables are not modifiable
- Scope is the body of the parent element

- **Examples:**
  ```xml
  <xsl:variable name="n" select="2+2"/>
  <foo bar="{$n}"/>
  
  <xsl:variable name="n">2</xsl:variable>
  <xsl:value-of select="item[position()=$n]"/>
  
  <xsl:variable name="x"><xsl:apply-templates/></xsl:variable>
  <xsl:copy-of select="$x"/>
  ```
XSLT vs. XQuery

• XSLT has in addition to XQuery:
  – Dynamically generated ns (2.0)
  – Positional grouping (2.0)
  – Various output methods (HTML, XML, TEXT)
  – Built-in dynamic dispatch

• Xquery has lots of extra features

• Both are Turing complete

• Some implementations use the same runtime for both languages, and a simple different language layer (same data model, same Xpath, same libraries)
  – E.g. Oracle, Saxon

• No standard way to cross-call between the two languages (future work)

• Optimization:
  – *Much easier* in Xquery because dataflow is possible. Not possible in XSLT (plus iterations are cheaper then recursive calls)
  – Same reason why Xquery has a static typing and XSLT doesn’t
When to use each one?

• It depends on the application
• Mix and match (see Pipelining in W3C)
• Document formatting is better done in XSLT
• Data-style, query-style computations are better done in XQuery
• “Depends on the degree of entropy in the data”
  – Irregular data => XSLT
  – Regular data => XQuery
• Any side-effecting requirements imply XQuery (no updates and side-effects in XSLT)
• More then one document => almost implies XQuery
• Performance will certainly be better in XQuery
• XQuery will be a richer language in the future
• XQuery widely present in universities, XSLT no
• Automatic translators between the two languages
Frequent solutions to XML data management

- Map it to *generic* programming APIs (e.g. DOM, SAX, StaX)
- *Manually* map it to *non-generic* APIs
- *Automatically* map it to *non-generic* structures
- Use *XML extensions* of existing languages
- Shredding for relational stores
- *Native XML processing through XSLT and XQuery*