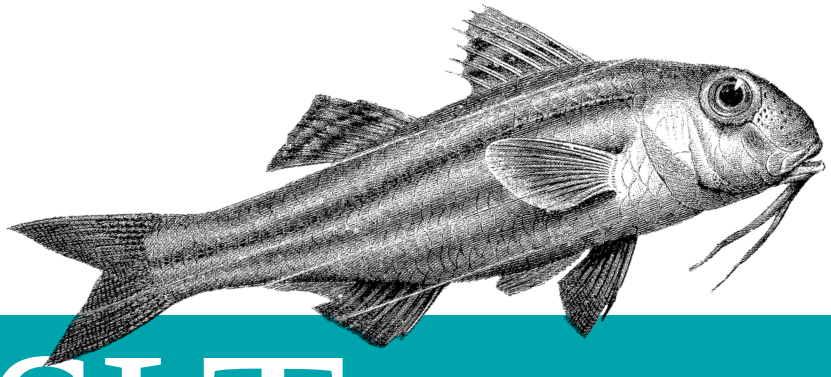
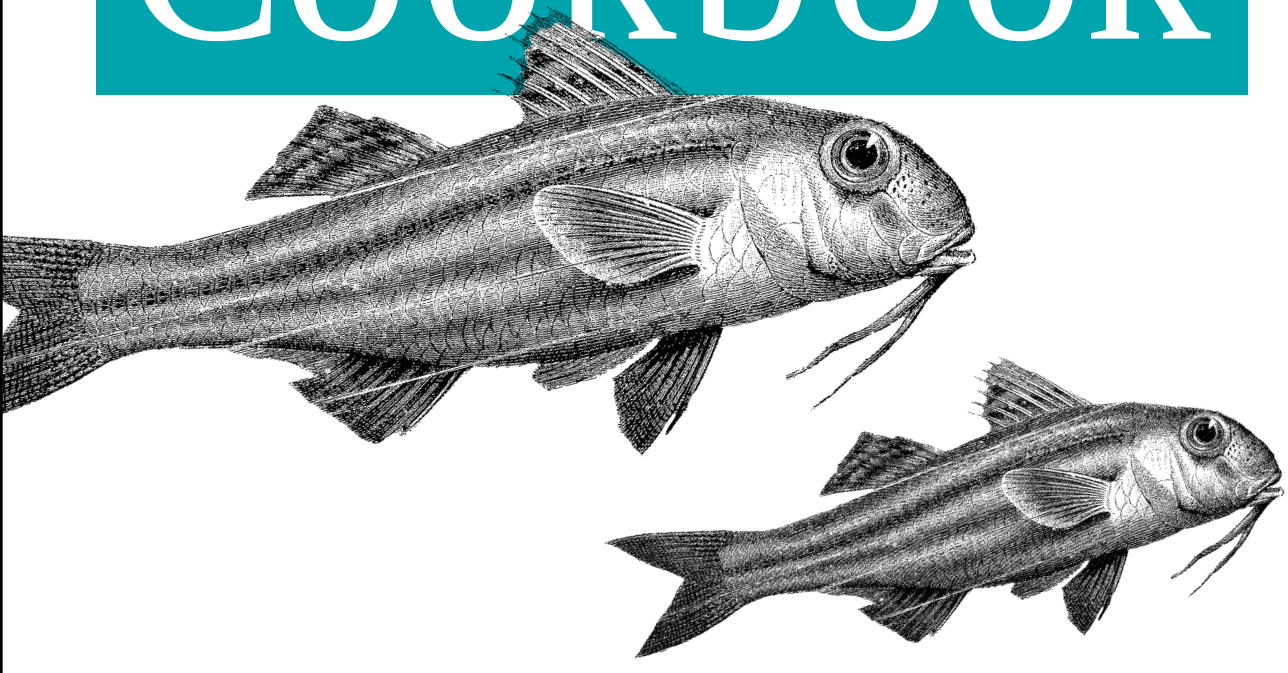
Solutions and Examples for XML and XSLT Developers



XSLT Cookbook



O'REILLY®

Sal Mangano

XSLT Cookbook

Sal Mangano

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 7
Querying XML

```
<xsl:template name="child-query">
<xsl:with-param name="parent" select=" 'Daddy' "/>
<xsl:value-of select="concat('But, why',$parent,'?')">
<xsl:apply-templates select="reasonable_response"/>
<xsl:call-template name="child-query">
<xsl:with-param name="parent" select="$parent"/>
</xsl:call-template>
</xsl:template>
```

—Parents not recognizing tail recursion
may risk blowing their stack

This chapter covers recipes for using XSLT as an XML query language. Querying XML means extracting information from one or more XML documents to answer questions about facts and relationships occurring in and among these documents. By analogy, querying an XML document involves asking the same types of questions of XML using XSLT that one might ask of a relational database using SQL.

The "official" query language for XML promulgated by the W3C is not XSLT, but XQuery (*http://www.w3.org/TR/xquery/*). XSLT and XQuery have many similarities, but also some striking differences. For example, XSLT and XQuery both rely on XPath. However, an XSLT script is always in XML syntax, while an XQuery script has both a human-friendly and XML syntax (*http://www.w3.org/TR/xqueryx*).

When the idea for an XML query language distinct from XSLT was proposed, it was controversial. Many members of the XML community thought there would be too much overlap between the two. Indeed, any query formulated in XQuery could also be implemented in XSLT. In many cases, the XSLT solution is as concise as the XQuery solution. The advantage of XQuery is that it is generally easier to understand than the equivalent XSLT. Indeed, XQuery should present a much smaller learning curve to those already versed in SQL. Obviously, comprehension is also a function of what you are used to, so these comparisons are not absolute.

Explaining XQuery in detail or providing a detailed comparison between it and XSLT is beyond the scope of this chapter. Instead, this chapter provides query examples for those who have already invested time into XSLT and do not wish to learn yet another XML-related language.

It would be impossible to create examples that exhausted all types of queries you might want to run on XML data. Instead, this chapter takes a two-pronged approach. First, it presents primitive and generally applicable query examples. These examples are building blocks that can be adapted to solve more complex query problems. Second, it presents a recipe that shows solutions to most XML query-use cases presented in the W3C document *XML Query Use Cases* (*http://www.w3.org/TR/xmlquery-use-cases*). In many cases, you can find a solution to a use-case instance that is similar enough to the particular query problem you face. It then becomes a simple matter of adapting the solution to the particulars of your XML data.

7.1 Performing Set Operations on Node Sets

Problem

You need to find the union, intersection, set difference, or symmetrical set difference between two node sets. You may also need to test equality and subset relationships between two node sets.

Solution

The union is trivial because XPath supports it directly:

```
<xsl:copy-of select="$node-set1 | $node-set2"/>
```

The intersection of two node sets requires a more convoluted expression:

```
<xsl:copy-of select="$node-set1[count(. | $node-set2) = count($node-set2)]"/>
```

This means all elements in `node-set1` that are also in `node-set2` by virtue of the fact that forming the union with `node-set2` and some specified element in `node-set1` leaves the same set of elements.

Set difference (those elements that are in the first set but not the second) follows:

```
<xsl:copy-of select="$node-set1[count(. | $node-set2) != count($node-set2)]"/>
```

This means all elements in `node-set1` that are not also in `node-set2` by virtue of the fact that forming the union with `node-set2` and some specified element in `node-set1` produces a set with more elements.

An example of symmetrical set difference (the elements are in one set but not the other) follows:

```
<xsl:copy-of select="$node-set1[count(. | $node-set2) != count($node-set2)] |
$node-set2[count(. | $node-set1) != count($node-set1)] "/>
```

The symmetrical set difference is simply the union of the differences taken both ways.

To test if `node-set1` is equal to `node-set2`:

```
<xsl:if test="count($ns1|$ns2) = count($ns1) and
              count($ns1) = count($ns2)">
```

Two sets are equal if their union produces a set with the same number of elements as are contained in both sets individually.

To test if `node-set1` is a subset of `node-set2`:

```
<xsl:if test="count($node-set1|$node-set2) = count($node-set1)">
```

To test if `node-set1` is a proper subset of `node-set2`:

```
<xsl:if test="count($ns1|$ns2) = count($ns1) and count($ns1) > count(ns2)">
```

Discussion

You may wonder what set operations have to do with XML queries. Set operations are ways of finding commonalities and differences between sets of elements extracted from a document. Many basic questions one can ask of data have to do with common and distinguishing traits.

For example, imagine extracting person elements from *people.xml* as follows:

```
<xsl:variable name="males" select="//person[@sex='m']"/>
<xsl:variable name="females" select="//person[@sex='f']"/>
<xsl:variable name="smokers" select="//person[@smoker='yes']"/>
<xsl:variable name="non-smokers" select="//person[@smoker='no']"/>
```

Now if you were issuing life insurance, you might consider charging each of the following sets of people different rates:

```
<!-- Male smokers -->
<xsl:variable name="super-risk"
     select="$males[count(. | $smokers) = count($smokers)]"/>
<!-- Female smokers -->
<xsl:variable name="high-risk"
     select="$females[count(. | $smokers) = count($smokers)]"/>
<!-- Male non-smokers -->
<xsl:variable name="moderate-risk"
     select="$males[count(. | $non-smokers) = count($non-smokers)]"/>
<!-- Female non-smokers -->
<xsl:variable name="low-risk"
     select="$females[count(. | $non-smokers) = count($non-smokers)]"/>
```

You probably noticed that the same answers could have been acquired more directly by using logic rather than set theory:

```
<!-- Male smokers -->
<xsl:variable name="super-risk"
     select="//person[@sex='m' and @smoker='y']"/>
<!-- Female smokers -->
```

```
<xsl:variable name="high-risk"
     select="//person[@sex='f' and @smoker='y']"/>
<!-- Male non-smokers -->
<xsl:variable name="moderate-risk"
     select="//person[@sex='m' and @smoker='n']"/>
<!-- Female non-smokers -->
<xsl:variable name="low-risk"
     select="//person[@sex='f' and @smoker='n']"/>
```

Better still, if you already had the set of males and females extracted, it would be more efficient to say:

```
<!-- Male smokers -->
<xsl:variable name="super-risk"
     select="$males[@smoker='y']"/>
<!-- Female smokers -->
<xsl:variable name="high-risk"
     select="$females[@smoker='y']"/>
<!-- Male non-smokers -->
<xsl:variable name="moderate-risk"
     select="$males[@smoker='n']"/>
<!-- Female non-smokers -->
<xsl:variable name="low-risk"
     select="$females[@smoker='n']"/>
```

These observations do not invalidate the utility of the set approach. Notice that the set operations work without knowledge of what the sets themselves contain. Set operations work at a higher level of abstraction. Imagine that you have a complex XML document and are interested in the following four sets:

```
<!-- All elements that have elements c1 or c2  as children-->
<xsl:variable name="set1" select="//*[c1 or c2]"/>
<!-- All elements that have elements c3 and c4  as children-->
<xsl:variable name="set2" select="//*[c3 and c4]"/>
<!-- All elements whose parent has attribute a1-->
<xsl:variable name="set3" select="//*[../@a1]"/>
<!-- All elements whose parent has attribute a2-->
<xsl:variable name="set4" select="//*[../@a2]"/>
```

In the original example, it was obvious that the sets of males and females (and smokers and nonsmokers) are disjoint. Here you have no such knowledge. The sets may be completely disjointed, completely overlap, or share only some elements. There are only two ways to find out what is in common between, say, `set1` and `set3`. The first is to take their intersection; the second is to traverse the entire document again using the logical `and` of their predicates. In this case, the intersection is clearly the way to go.

EXSLT defines a set module that includes functions performing the set operations discussed here. The EXSLT uses an interesting technique to return the result of its set operations. Instead of returning the result directly, it applies templates to the result in a mode particular to the type of set operation. For example, after EXSLT `set:intersection` computes the intersection, it invokes `<xsl:apply-templates`

mode="set:intersection"/> on the result. A default template exists in EXSLT with this mode, and it will return a copy of the result as a node-tree fragment. This indirect means of returning the result allows users importing the EXSLT set module to override the default to process it further. This technique is useful but limited. It is useful because it potentially eliminates the need to use the node-set extension function to convert the result back into a node set. It is limited because there can be at most one such overriding template per matching pattern in the user stylesheet for each operation. However, you may want to do very different post-processing tasks with the result of intersections invoked from different places in the same stylesheet.

Do not be alarmed if you do not grasp the subtleties of EXSLT's technique discussed here. Chapter 14 will discuss in more detail these and other techniques for making XSLT code reusable.

See Also

You can find an explanation of the EXSLT set operations at *http://www.exslt.org/set/index.html*.

7.2 Performing Set Operations on Node Sets Using Value Semantics

Problem

You need to find the union, intersection, set difference, or symmetrical set difference between elements in two node sets; however, in your problem, *equality* is not defined as *node-set identity*. In other words, *equality* is a function of a node's value.

Solution

The need for this solution may arise when working with multiple documents. Consider two documents with the same DTD but content that may not contain duplicate element values. XSLT elements coming from distinct documents are distinct even if they contain elements with the same namespace, attribute, and text values. See Examples 7-1 to 7-4.

Example 7-1. people1.xslt

```
<people>
  <person name="Brad York" age="38" sex="m" smoker="yes"/>
  <person name="Charles Xavier" age="32" sex="m" smoker="no"/>
  <person name="David Willimas" age="33" sex="m" smoker="no"/>
</people>
```

Example 7-2. people2.xslt

```
<people>
  <person name="Al Zehtooney" age="33" sex="m" smoker="no"/>
  <person name="Brad York" age="38" sex="m" smoker="yes"/>
  <person name="Charles Xavier" age="32" sex="m" smoker="no"/>
</people>
```

Example 7-3. Failed attempt to use XSLT union to select unique people

```
<xsl:template match="/">
  <people>
    <xsl:copy-of select="//person | document('people2.xml')//person"/>
  </people>
</xsl:template>
```

Example 7-4. Output when run with people1.xml as input

```
<people>
   <person name="Brad York" age="38" sex="m" smoker="yes"/>
   <person name="Charles Xavier" age="32" sex="m" smoker="no"/>
   <person name="David Willimas" age="33" sex="m" smoker="no"/>
   <person name="Al Zehtooney" age="33" sex="m" smoker="no"/>
   <person name="Brad York" age="38" sex="m" smoker="yes"/>
   <person name="Charles Xavier" age="32" sex="m" smoker="no"/>
</people>
```

Relying on node identity can also break down in single document cases when you want equality of nodes to be a function of their text or attribute values.

The following stylesheet provides a reusable implementation of union, intersection, and set difference based on value semantics. The idea is that a stylesheet importing this one will override the template whose `mode=" vset:element-equality"`. This allows the importing stylesheet to define whatever equality semantics make sense for the given input:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:vset="http:/www.ora.com/XSLTCookbook/namespaces/vset">

<xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

<!-- The default implementation of element equality. Override in the importing
stylesheet as neccessary. -->
<xsl:template match="node( ) | @*" mode="vset:element-equality">
  <xsl:param name="other"/>
  <xsl:if test=". = $other">
    <xsl:value-of select="true( )"/>
  </xsl:if>
</xsl:template>

<!-- The default set membership test uses element equality. You will rarely need to
override this in the importing stylesheet. -->
<xsl:template match="node( ) | @*" mode="vset:member-of">
  <xsl:param name="elem"/>
```

```
  <xsl:variable name="member-of">
    <xsl:for-each select=".">
      <xsl:apply-templates select="." mode="vset:element-equality">
        <xsl:with-param name="other" select="$elem"/>
      </xsl:apply-templates>
    </xsl:for-each>
  </xsl:variable>
  <xsl:value-of select="string($member-of)"/>
</xsl:template>

<!-- Compute the union of two sets using "by value" equality. -->
<xsl:template name="vset:union">
  <xsl:param name="nodes1" select="/.." />
  <xsl:param name="nodes2" select="/.." />
  <!-- for internal use -->
  <xsl:param name="nodes" select="$nodes1 | $nodes2" />
  <xsl:param name="union" select="/.." />
  <xsl:choose>
    <xsl:when test="$nodes">
      <xsl:variable name="test">
        <xsl:apply-templates select="$union" mode="vset:member-of">
          <xsl:with-param name="elem" select="$nodes[1]" />
        </xsl:apply-templates>
      </xsl:variable>
      <xsl:call-template name="vset:union">
        <xsl:with-param name="nodes" select="$nodes[position( ) > 1]" />
        <xsl:with-param name="union"
                        select="$union | $nodes[1][not(string($test))]" />
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-templates select="$union" mode="vset:union" />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- Return a copy of union by default. Override in importing stylesheet  to recieve
reults as a "callback"-->
<xsl:template match="/ | node( ) | @*" mode="vset:union">
  <xsl:copy-of select="."/>
</xsl:template>

<!-- Compute the intersection of two sets using "by value" equality. -->
<xsl:template name="vset:intersection">
  <xsl:param name="nodes1" select="/.."/>
  <xsl:param name="nodes2" select="/.."/>
  <!-- For internal use -->
  <xsl:param name="intersect" select="/.."/>

  <xsl:choose>
    <xsl:when test="not($nodes1)">
      <xsl:apply-templates select="$intersect" mode="vset:intersection"/>
    </xsl:when>
    <xsl:when test="not($nodes2)">
```

```
      <xsl:apply-templates select="$intersect" mode="vset:intersection"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="test1">
        <xsl:apply-templates select="$nodes2" mode="vset:member-of">
          <xsl:with-param name="elem" select="$nodes1[1]"/>
        </xsl:apply-templates>
      </xsl:variable>
      <xsl:variable name="test2">
        <xsl:apply-templates select="$intersect" mode="vset:member-of">
          <xsl:with-param name="elem" select="$nodes1[1]"/>
        </xsl:apply-templates>
      </xsl:variable>
      <xsl:choose>
        <xsl:when test="string($test1) and not(string($test2))">
          <xsl:call-template name="vset:intersection">
            <xsl:with-param name="nodes1"
                    select="$nodes1[position( ) > 1]"/>
            <xsl:with-param name="nodes2" select="$nodes2"/>
            <xsl:with-param name="intersect"
                    select="$intersect | $nodes1[1]"/>
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <xsl:call-template name="vset:intersection">
            <xsl:with-param name="nodes1"
                    select="$nodes1[position( ) > 1]"/>
            <xsl:with-param name="nodes2" select="$nodes2"/>
            <xsl:with-param name="intersect" select="$intersect"/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- Return a copy of intersection by default. Override in importing stylesheet to
recieve results as a "callback"-->
<xsl:template match="/ | node( ) | @*" mode="vset:intersection">
  <xsl:copy-of select="."/>
</xsl:template>

<!-- Compute the differnce between two sets (node1 - nodes2) using "by value"
equality. -->
<xsl:template name="vset:difference">
  <xsl:param name="nodes1" select="/.."/>
  <xsl:param name="nodes2" select="/.."/>
  <!-- For internal use -->
  <xsl:param name="difference" select="/.."/>

  <xsl:choose>
    <xsl:when test="not($nodes1)">
      <xsl:apply-templates select="$difference" mode="vset:difference"/>
    </xsl:when>
```

```
    <xsl:when test="not($nodes2)">
      <xsl:apply-templates select="$nodes1" mode="vset:difference"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="test1">
        <xsl:apply-templates select="$nodes2" mode="vset:member-of">
          <xsl:with-param name="elem" select="$nodes1[1]"/>
        </xsl:apply-templates>
      </xsl:variable>
      <xsl:variable name="test2">
        <xsl:apply-templates select="$difference" mode="vset:member-of">
          <xsl:with-param name="elem" select="$nodes1[1]"/>
        </xsl:apply-templates>
      </xsl:variable>
      <xsl:choose>
        <xsl:when test="string($test1) or string($test2)">
          <xsl:call-template name="vset:difference">
            <xsl:with-param name="nodes1"
                    select="$nodes1[position( ) > 1]"/>
            <xsl:with-param name="nodes2" select="$nodes2"/>
            <xsl:with-param name="difference" select="$difference"/>
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <xsl:call-template name="vset:difference">
            <xsl:with-param name="nodes1"
                    select="$nodes1[position( ) > 1]"/>
            <xsl:with-param name="nodes2" select="$nodes2"/>
            <xsl:with-param name="difference"
                    select="$difference | $nodes1[1]"/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- Return a copy of difference by default. Override in importing stylesheet to
recieve results as a "callback"-->
<xsl:template match="/ | node( ) | @*" mode="vset:difference">
  <xsl:copy-of select="."/>
</xsl:template>
```

These recursive templates are implemented in terms of the following definitions:

*Union(*nodes1,nodes2*)*
: The union includes everything in nodes2 plus everything in nodes1 not already a member of nodes2.

*Intersection(*nodes1,nodes2*)*
: The intersection includes everything in nodes1 that is also a member of nodes2.

*Difference(*nodes1,nodes2*)*
: The difference includes everything in nodes1 that is not also a member of nodes2.

In all cases, membership defaults to equality of string values, but the importing stylesheet can override this default.

Given these value-oriented set operations, you can achieve the desired effect on *people1.xml* and *people2.xml* using the following stylesheet:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 xmlns:vset="http:/www.ora.com/XSLTCookbook/namespaces/vset">

<xsl:import href="set.ops.xslt"/>

<xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

<xsl:template match="/">
  <people>
    <xsl:call-template name="vset:union">
      <xsl:with-param name="nodes1" select="//person"/>
      <xsl:with-param name="nodes2" select="document('people2.xml')//person"/>
    </xsl:call-template>
  </people>
</xsl:template>

<!--Define person equality as having the same name -->
<xsl:template match="person" mode="vset:element-equality">
  <xsl:param name="other"/>
  <xsl:if test="@name = $other/@name">
    <xsl:value-of select="true( )"/>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Discussion

You might think that equality is a cut-and-dried issue; two things are either equal or they're not. However, in programming (as in politics), equality is in the eye of the beholder. In a typical document, an element is associated with a uniquely identifiable object. For example, a paragraph element, `<p>...</p>`, is distinct from another paragraph element somewhere else in the document, even if they have the same content. Hence, set operations based on the unique identity of elements are the norm. However, when considering XSLT operations crossing multiple documents or acting on elements that result from applying `xsl:copy`, we need to carefully consider what we want equality to be.

Here are some query examples in which value set semantics are required:

1. You have two documents from different namespaces. Examples 7-5 to 7-8 help you find all the element (local) names these documents have in common and those that are unique to each namespace.

Example 7-5. doc1.xml

```
<doc xmlns:doc1="doc1" xmlns="doc1">
  <chapter number="1">
    <section number="1">
      <p>
        Once upon a time...
      </p>
    </section>
  </chapter>
  <chapter number="2">
    <note to="editor">I am still waiting for my $100000 advance.</note>
    <section number="1">
      <p>
        ... and they lived happily ever after.
      </p>
    </section>
  </chapter>
</doc>
```

Example 7-6. doc2.xml

```
<doc xmlns:doc1="doc2" xmlns="doc2">
  <chapter number="1">
    <section number="1">
      <sub>
        <p>
          Once upon a time...
          <ref type="footnote" number="1"/>
        </p>
      </sub>
      <fig>Figure1</fig>
    </section>
    <footnote number="1">
      Hey diddle diddle.
    </footnote>
  </chapter>
  <chapter number="2">
    <section number="1">
      <p>
        ... and they lived happily ever after.
      </p>
    </section>
  </chapter>
</doc>
```

Example 7-7. unique-element-names.xslt

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:doc1="doc1" xmlns:doc2="doc2"
  xmlns:vset="http:/www.ora.com/XSLTCookbook/namespaces/vset"
  extension-element-prefixes="vset">

  <xsl:import href="set.ops.xslt"/>
```

Example 7-7. unique-element-names.xslt (continued)

```
  <xsl:output method="text" />

  <xsl:template match="/">
    <xsl:text>&#xa;The elements in common are: </xsl:text>
    <xsl:call-template name="vset:intersection">
      <xsl:with-param name="nodes1" select="//*"/>
      <xsl:with-param name="nodes2" select="document('doc2.xml')//*"/>
    </xsl:call-template>

    <xsl:text>&#xa;The elements only in doc1 are: </xsl:text>
    <xsl:call-template name="vset:difference">
      <xsl:with-param name="nodes1" select="//*"/>
      <xsl:with-param name="nodes2" select="document('doc2.xml')//*"/>
    </xsl:call-template>

    <xsl:text>&#xa;The elements only in doc2 are: </xsl:text>
    <xsl:call-template name="vset:difference">
      <xsl:with-param name="nodes1" select="document('doc2.xml')//*"/>
      <xsl:with-param name="nodes2" select="//*"/>
    </xsl:call-template>
    <xsl:text>&#xa;</xsl:text>

  </xsl:template>

  <xsl:template match="*" mode="vset:intersection">
     <xsl:value-of select="local-name(.)"/>
     <xsl:if test="position( ) != last( )">
       <xsl:text>, </xsl:text>
     </xsl:if>
  </xsl:template>

  <xsl:template match="*" mode="vset:difference">
     <xsl:value-of select="local-name(.)"/>
     <xsl:if test="position( ) != last( )">
       <xsl:text>, </xsl:text>
     </xsl:if>
  </xsl:template>

  <xsl:template match="doc1:* | doc2:*" mode="vset:element-equality">
   <xsl:param name="other"/>
    <xsl:if test="local-name(.) = local-name($other)">
      <xsl:value-of select="true( )"/>
    </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

Example 7-8. Output

```
The elements in common are: doc, chapter, section, p
The elements only in doc1 are: note
The elements only in doc2 are: sub, ref, fig, footnote
```

2. A Visio XML document consists of master shapes, master-shape instances, and user-defined shapes with no corresponding master. You would like to extract the data for all unique shapes. For purpose of this query, two shapes are equal if either of the following are true:
   a. They both have master attributes, `@Master`, and these attribute values are equal.
   b. At least one lacks a master attribute, but their geometry elements, `Geom`, are equal. Geometry elements are equal if all attributes of all descendants of `Geom` are equal.

   Otherwise, they are not equal.

   This query can be implemented by taking the intersection of the set all shapes with itself under the rules of equality stated earlier.* You can also use the `vset:union` template with the nodes parameter:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:vxd="urn:schemas-microsoft-com:office:visio"
  xmlns:vset="http:/www.ora.com/XSLTCookbook/namespaces/vset"
  extension-element-prefixes="vset">

  <xsl:import href="set.ops.xslt"/>

  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

  <xsl:template match="/">
  <UniqueShapes>
      <xsl:call-template name="vset:intersection">
        <xsl:with-param name="nodes1" select="//vxd:Pages/*/*/vxd:Shape"/>
        <xsl:with-param name="nodes2" select="//vxd:Pages/*/*/vxd:Shape"/>
      </xsl:call-template>
    </UniqueShapes>
  </xsl:template>

  <xsl:template match="vxd:Shape" mode="vset:intersection">
    <xsl:copy-of select="." />
  </xsl:template>

<xsl:template match="vxd:Shape" mode="vset:element-equality">
  <xsl:param name="other"/>
  <xsl:choose>
    <xsl:when test="@Master and $other/@Master and @Master = $other/@Master">
      <xsl:value-of select="true( )"/>
    </xsl:when>
    <xsl:when test="not(@Master) or not($other/@Master)">
      <xsl:variable name="geom1">
```

* A mathematician will tell you that the intersection of a set with itself will always yield the same set. This is true for proper sets (with no duplicates). However, here you are using an application-specific notion of equality, and the node sets typically will not be proper sets under that equality test. However, the value-set operations always produce proper sets, so this technique is a way of removing duplicates.

```
        <xsl:for-each select="vxd:Geom//*/@*">
          <xsl:sort select="name( )"/>
          <xsl:value-of select="."/>
        </xsl:for-each>
      </xsl:variable>
       <xsl:variable name="geom2">
        <xsl:for-each select="$other/vxd:Geom//*/@*">
          <xsl:sort select="name( )"/>
          <xsl:value-of select="."/>
        </xsl:for-each>
      </xsl:variable>
       <xsl:if test="$geom1 = $geom2">
        <xsl:value-of select="true( )"/>
       </xsl:if>
     </xsl:when>
   </xsl:choose>
 </xsl:template>


 </xsl:stylesheet>
```

7.3 Determining Set Equality by Value

Problem

You need to determine if the nodes in one node set are equal (by value) to the nodes in another node set (ignoring order).

Solution

This problem is slightly more subtle than it appears on the surface. Consider an obvious solution that works in many cases:

```
<xsl:template name="vset:equal-text-values">
  <xsl:param name="nodes1" select="/.."/>
  <xsl:param name="nodes2" select="/.."/>
  <xsl:choose>
   <!--Empty node-sets have equal values -->
    <xsl:when test="not($nodes1) and not($nodes2)">
      <xsl:value-of select="true( )"/>
      </xsl:when>
    <!--Node sets of unequal sizes can not have equal values -->
    <xsl:when test="count($nodes1) != count($nodes2)"/>
    <!--If an element of nodes1 is present in nodes2 then the node sets
     have equal values if the node sets without the common element have equal
     values -->
    <xsl:when test="$nodes1[1] = $nodes2">
      <xsl:call-template name="vset:equal-text-values">
          <xsl:with-param name="nodes1" select="$nodes1[position( )>1]"/>
          <xsl:with-param name="nodes2"
                         select="$nodes2[not(. = $nodes1[1])]"/>
```

```
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise/>
  </xsl:choose>
</xsl:template>
```

We have chosen a name for this equality test to emphasize the context in which it should be applied. That is when value equality indicate string-value equality. Clearly, this template will not give the correct result if equality is based on attributes or criteria that are more complex. However, this template has a more subtle problem. It tacitly assumes that the compared node sets are proper sets (i.e., they contain no duplicates) under string-value equality. In some circumstances, this may not be the case. Consider the following XML that represents the individuals who borrowed books from a library:

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
  <book>
    <name>High performance Java programming.</name>
    <borrowers>
      <borrower>James Straub</borower>
    </borrowers>
  </book>
  <book>
    <name>Exceptional C++</name>
    <borrowers>
      <borrower>Steven Levitt</borower>
    </borrowers>
  </book>
  <book>
    <name>Design Patterns</name>
    <borrowers>
      <borrower>Steven Levitt</borower>
      <borrower>James Straub</borower>
      <borrower>Steven Levitt</borower>
    </borrowers>
  </book>
  <book>
    <name>The C++ Programming Language</name>
    <borrowers>
      <borrower>James Straub</borower>
      <borrower>James Straub</borower>
      <borrower>Steven Levitt</borower>
    </borrowers>
  </book>
</library>
```

If an individual's name appears more than once, it simply means he borrowed the book more than once. Now, if you wrote a query to determine all books borrowed by the same people, most would agree that *Design Patterns* and *The C++ Programming Language* qualify as two such books. However, if you used `vset:equal-text-values` in the implementation of that query, you would not get this result because it

assumes that sets do not contain duplicates. You can alter `vset:equal-text-values` to tolerate duplicates with the following changes:

```
<xsl:template name="vset:equal-text-values-ignore-dups">
  <xsl:param name="nodes1" select="/.."/>
  <xsl:param name="nodes2" select="/.."/>
  <xsl:choose>
   <!--Empty node-sets have equal values -->
    <xsl:when test="not($nodes1) and not($nodes2)">
      <xsl:value-of select="true( )"/>
      </xsl:when>
    <!--If an element of nodes1 is present in nodes2 then the node sets
     have equal values if the node sets without the common element have equal
     values -->
    <!--delete this line
          <xsl:when test="count($nodes1) != count($nodes2)"/> -->
    <xsl:when test="$nodes1[1] = $nodes2">
      <xsl:call-template name="vset:equal-text-values">
          <xsl:with-param name="nodes1"
                          select="$nodes1[not(. = $nodes1[1])]"/>
          <xsl:with-param name="nodes2"
               select="$nodes2[not(. = $nodes1[1])]"/>      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise/>
  </xsl:choose>
</xsl:template>
```

Notice that we have commented out the test for unequal sizes because that test is not valid in the presence of duplicates. For example, one set might have three occurrences of an element with string value `foo`, while the other has a single element `foo`. These sets should be equal when duplicates are ignored. You also must do more than remove just the first element on the recursive step; you should remove all elements with the same value as the first element, just as you do for the second set. This will ensure that duplicates are fully accounted for on each recursive pass. These changes make all equality tests based on text value come out correct, but at the cost of doing additional work on sets that are obviously unequal.

These equality tests are not as general as the value-set operations produced in Recipe 7.2 because they presume that the only notion of equality you care about is text-value equality. You can generalize them by reusing the same technique you used for testing membership based on a test of element equality that can be overridden by an importing stylesheet:

```
<xsl:template name="vset:equal">
  <xsl:param name="nodes1" select="/.."/>
  <xsl:param name="nodes2" select="/.."/>
  <xsl:if test="count($nodes1) = count($nodes2)">
    <xsl:call-template name="vset:equal-impl">
      <xsl:with-param name="nodes1" select="$nodes1"/>
      <xsl:with-param name="nodes2" select="$nodes2"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
```

```
<!-- Once we know the sets have the same number of elements -->
<!-- we only need to test that every member of the first set is -->
<!-- a member of the second -->
<xsl:template name="vset:equal-impl">
  <xsl:param name="nodes1" select="/.."/>
  <xsl:param name="nodes2" select="/.."/>
  <xsl:choose>
    <xsl:when test="not($nodes1)">
      <xsl:value-of select="true( )"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="test">
        <xsl:apply-templates select="$nodes2" mode="vset:member-of">
          <xsl:with-param name="elem" select="$nodes1[1]"/>
        </xsl:apply-templates>
      </xsl:variable>
      <xsl:if test="string($test)">
        <xsl:call-template name="vset:equal-impl">
          <xsl:with-param name="nodes1" select="$nodes1[position( ) > 1]"/>
          <xsl:with-param name="nodes2" select="$nodes2"/>
        </xsl:call-template>
      </xsl:if>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

If you want generalized equality that works in the presence of duplicates, then you must apply a more brute-force approach that makes two passes over the sets:

```
<xsl:template name="vset:equal-ignore-dups">
  <xsl:param name="nodes1" select="/.."/>
  <xsl:param name="nodes2" select="/.."/>

  <xsl:variable name="mismatch1">
    <xsl:for-each select="$nodes1">
      <xsl:variable name="test-elem">
        <xsl:apply-templates select="$nodes2" mode="vset:member-of">
          <xsl:with-param name="elem" select="."/>
        </xsl:apply-templates>
      </xsl:variable>
      <xsl:if test="not(string($test-elem))">
        <xsl:value-of select=" 'false' "/>
      </xsl:if>
    </xsl:for-each>
  </xsl:variable>
  <xsl:if test="not($mismatch1)">
    <xsl:variable name="mismatch2">
      <xsl:for-each select="$nodes2">
        <xsl:variable name="test-elem">
          <xsl:apply-templates select="$nodes1" mode="vset:member-of">
            <xsl:with-param name="elem" select="."/>
          </xsl:apply-templates>
        </xsl:variable>
```

```
          <xsl:if test="not(string($test-elem))">
            <xsl:value-of select=" 'false' "/>
          </xsl:if>
        </xsl:for-each>
      </xsl:variable>
      <xsl:if test="not($mismatch2)">
        <xsl:value-of select="true( )"/>
      </xsl:if>
    </xsl:if>
  </xsl:template>
```

This template works by iterating over the first set and looking for elements that are not a member of the second. If no such element is found, the variable `$mismatch1` will be null. In that case, it must repeat the test in the other direction by iterating over the second set.

Discussion

The need to test set equality comes up often in queries. Consider the following tasks:

- Find all books having the same authors.
- Find all suppliers who stock the same set of parts.
- Find all families with same-age children.

Whenever you encounter a one-to-many relationship and you are interested in elements that have the same set of associated elements, the need to test set equality will arise.

7.4 Performing Structure-Preserving Queries

Problem

You need to query an XML document so that the response has a structure that is identical to the original.

Solution

Structure-preserving queries filter out irrelevant information while preserving most of the document structure. The degree by which the output structure resembles the structure of the input is the metric that determines the applicability of this example. The more similar it is, the more this example applies.

The example has two components—one reusable and the other custom. The reusable component is a stylesheet that copies all nodes to the output (identity transform). We used this stylesheet, shown in Example 7-9, extensively in Chapter 6.

Example 7-9. copy.xslt

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/ | node( ) | @*">
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

The custom component is a stylesheet that imports *copy.xslt* and creates rules to override its default behavior. For example, the following stylesheet results in output identical to *people.xml*, but with only female smokers:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

     <xsl:import href="copy.xslt"/>

    <!-- Collapse space left by removing person elements -->
    <xsl:strip-space elements="people"/>

     <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

     <xsl:template match="person[@sex = 'f' and @smoker='yes']">
       <!-- Apply default behavior, which is to copy -->
       <xsl:apply-imports/>
     </xsl:template>

     <!-- Ignore other people -->
     <xsl:template match="person"/>

</xsl:stylesheet>
```

Alternatively, a single template can match the things that you want to exclude and do nothing with them:

```
<xsl:template match="person[@sex != 'f' or @smoker != 'yes']" />
```

Discussion

This example is extremely useful because it lets you preserve the structure of an XML document without necessarily knowing what its structure is. You only need to know what elements should be filtered out and that you create templates that do so.

This example is applicable in contexts that most people would not describe as queries. For example, suppose you wanted to clone an XML document, but remove all attributes named sex and replace them with an attribute called gender:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

     <xsl:import href="copy.xslt"/>

     <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

     <xsl:template match="@sex">
       <xsl:attribute name="gender">
          <xsl:value-of select="."/>
       </xsl:attribute>
     </xsl:template>

</xsl:stylesheet>
```

The beauty of this example is that it works on any XML document, regardless of its schema. If the document has elements with an attribute named sex, they will become gender:

Can you guess what the following variation does?*

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

     <xsl:import href="copy.xslt"/>

     <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

     <xsl:template match="@sex">
       <xsl:attribute name="gender">
          <xsl:value-of select="."/>
       </xsl:attribute>
     <xsl:apply-imports/>
     </xsl:template>

</xsl:stylesheet>
```

7.5 Joins

Problem

You want to relate elements in a document to other elements in the same or different document.

Solution

A *join* is the process of considering all pairs of element as being related (i.e., a Cartesian product) and keeping only those pairs that meet the join relationship (usually equality).

* It outputs both gender and sex attributes, but you knew that already!

To demonstrate, I have adapted the supplier parts database found in Date's *An Introduction to Database Systems* (Addison Wesley, 1986) to XML:

```
<database>
  <suppliers>
    <supplier id="S1" name="Smith" status="20" city="London"/>
    <supplier id="S2" name="Jones" status="10" city="Paris"/>
    <supplier id="S3" name="Blake" status="30" city="Paris"/>
    <supplier id="S4" name="Clark" status="20" city="London"/>
    <supplier id="S5" name="Adams" status="30" city="Athens"/>
  </suppliers>
  <parts>
    <part id="P1" name="Nut" color="Red" weight="12" city="London"/>
    <part id="P2" name="Bult" color="Green" weight="17" city="Paris"/>
    <part id="P3" name="Screw" color="Blue" weight="17" city="Rome"/>
    <part id="P4" name="Screw" color="Red" weight="14" city="London"/>
    <part id="P5" name="Cam" color="Blue" weight="12" city="Paris"/>
    <part id="P6" name="Cog" color="Red" weight="19" city="London"/>
  </parts>
  <inventory>
    <invrec sid="S1" pid="P1" qty="300"/>
    <invrec sid="S1" pid="P2" qty="200"/>
    <invrec sid="S1" pid="P3" qty="400"/>
    <invrec sid="S1" pid="P4" qty="200"/>
    <invrec sid="S1" pid="P5" qty="100"/>
    <invrec sid="S1" pid="P6" qty="100"/>
    <invrec sid="S2" pid="P1" qty="300"/>
    <invrec sid="S2" pid="P2" qty="400"/>
    <invrec sid="S3" pid="P2" qty="200"/>
    <invrec sid="S4" pid="P2" qty="200"/>
    <invrec sid="S4" pid="P4" qty="300"/>
    <invrec sid="S4" pid="P5" qty="400"/>
  </inventory>
</database>
```

The join to be performed will answer the question, "Which suppliers and parts are in the same city (co-located)?"

You can use two basic techniques to approach this problem in XSLT. The first uses nested `for-each` loops:

```
<xsl:template match="/">
  <result>
    <xsl:for-each select="database/suppliers/*">
      <xsl:variable name="supplier" select="."/>
      <xsl:for-each select="/database/parts/*[@city=current( )/@city]">
      <colocated>
        <xsl:copy-of select="$supplier"/>
        <xsl:copy-of select="."/>
      </colocated>
      </xsl:for-each>
    </xsl:for-each>
  </result>
</xsl:template>
```

The second approach uses apply-templates:

```
<xsl:template match="/">
  <result>
    <xsl:apply-templates select="database/suppliers/supplier" />
  </result>
</xsl:template>

<xsl:template match="supplier">
  <xsl:apply-templates select="/database/parts/part[@city = current( )/@city]">
    <xsl:with-param name="supplier" select="." />
  </xsl:apply-templates>
</xsl:template>

<xsl:template match="part">
  <xsl:param name="supplier" select="/.." />
  <colocated>
    <xsl:copy-of select="$supplier" />
    <xsl:copy-of select="." />
  </colocated>
</xsl:template>
```

If one of the sets of elements to be joined has a large number of members, then consider using xsl:key to improve performance:

```
<xsl:key name="part-city" match="part" use="@city"/>

<xsl:template match="/">
  <result>
    <xsl:for-each select="database/suppliers/*">
      <xsl:variable name="supplier" select="."/>
      <xsl:for-each select="key('part-city',$supplier/@city)">
      <colocated>
        <xsl:copy-of select="$supplier"/>
        <xsl:copy-of select="."/>
      </colocated>
      </xsl:for-each>
    </xsl:for-each>
  </result>
</xsl:template>
```

Each stylesheet produces the same result:

```
<result>
   <colocated>
      <supplier id="S1" name="Smith" status="20" city="London"/>
      <part id="P1" name="Nut" color="Red" weight="12" city="London"/>
   </colocated>
   <colocated>
      <supplier id="S1" name="Smith" status="20" city="London"/>
      <part id="P4" name="Screw" color="Red" weight="14" city="London"/>
   </colocated>
   <colocated>
      <supplier id="S1" name="Smith" status="20" city="London"/>
      <part id="P6" name="Cog" color="Red" weight="19" city="London"/>
   </colocated>
```

```
  <colocated>
    <supplier id="S2" name="Jones" status="10" city="Paris"/>
    <part id="P2" name="Bult" color="Green" weight="17" city="Paris"/>
  </colocated>
  <colocated>
    <supplier id="S2" name="Jones" status="10" city="Paris"/>
    <part id="P5" name="Cam" color="Blue" weight="12" city="Paris"/>
  </colocated>
  <colocated>
    <supplier id="S3" name="Blake" status="30" city="Paris"/>
    <part id="P2" name="Bult" color="Green" weight="17" city="Paris"/>
  </colocated>
  <colocated>
    <supplier id="S3" name="Blake" status="30" city="Paris"/>
    <part id="P5" name="Cam" color="Blue" weight="12" city="Paris"/>
  </colocated>
  <colocated>
    <supplier id="S4" name="Clark" status="20" city="London"/>
    <part id="P1" name="Nut" color="Red" weight="12" city="London"/>
  </colocated>
  <colocated>
    <supplier id="S4" name="Clark" status="20" city="London"/>
    <part id="P4" name="Screw" color="Red" weight="14" city="London"/>
  </colocated>
  <colocated>
    <supplier id="S4" name="Clark" status="20" city="London"/>
    <part id="P6" name="Cog" color="Red" weight="19" city="London"/>
  </colocated>
</result>
```

Discussion

The join you performed is called an *equi-join* because the elements are related by equality. More generally, joins can be formed using other relations. For example, consider the query, "Select all combinations of supplier and part information for which the supplier city follows the part city in alphabetical order."

It would be nice if you could simply write the following stylesheet, but XSLT 1.0 does not define relational operations on string types:

```
<xsl:template match="/">
  <result>
    <xsl:for-each select="database/suppliers/*">
      <xsl:variable name="supplier" select="."/>
       <!-- This does not work! -->
      <xsl:for-each select="/database/parts/*[current( )/@city > @city]">
      <colocated>
        <xsl:copy-of select="$supplier"/>
        <xsl:copy-of select="."/>
      </colocated>
      </xsl:for-each>
    </xsl:for-each>
  </result>
</xsl:template>
```

Instead, you must create a table using xsl:sort that can map city names onto integers that reflect the ordering. Here you rely on Saxon's ability to treat variables containing result-tree fragments as node sets when the version is set to 1.1. However, you can also use the node-set function of your particular XSLT 1.0 processor or use an XSLT 2.0 processor:

```
<xsl:stylesheet version="1.1" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
     <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

<xsl:variable name="unique-cities"
     select="//@city[not(. = ../preceding::*/@city)]"/>

<xsl:variable name="city-ordering">
  <xsl:for-each select="$unique-cities">
    <xsl:sort select="."/>
    <city name="{.}" order="{position( )}"/>
  </xsl:for-each>
</xsl:variable>

<xsl:template match="/">
  <result>
    <xsl:for-each select="database/suppliers/*">
      <xsl:variable name="s" select="."/>
      <xsl:for-each select="/database/parts/*">
        <xsl:variable name="p" select="."/>
        <xsl:if
          test="$city-ordering/*[@name = $s/@city]/@order &gt;
               $city-ordering/*[@name = $p/@city]/@order">
          <supplier-city-follows-part-city>
            <xsl:copy-of select="$s"/>
            <xsl:copy-of select="$p"/>
          </supplier-city-follows-part-city>
        </xsl:if>
      </xsl:for-each>
    </xsl:for-each>
  </result>
</xsl:template>

</xsl:stylesheet>
```

This query results in the following output:

```
<result>
   <supplier-city-follows-part-city>
      <supplier id="S2" name="Jones" status="10" city="Paris"/>
      <part id="P1" name="Nut" color="Red" weight="12" city="London"/>
   </supplier-city-follows-part-city>
   <supplier-city-follows-part-city>
      <supplier id="S2" name="Jones" status="10" city="Paris"/>
      <part id="P4" name="Screw" color="Red" weight="14" city="London"/>
   </supplier-city-follows-part-city>
   <supplier-city-follows-part-city>
      <supplier id="S2" name="Jones" status="10" city="Paris"/>
      <part id="P6" name="Cog" color="Red" weight="19" city="London"/>
   </supplier-city-follows-part-city>
```

```
    <supplier-city-follows-part-city>
       <supplier id="S3" name="Blake" status="30" city="Paris"/>
       <part id="P1" name="Nut" color="Red" weight="12" city="London"/>
    </supplier-city-follows-part-city>
    <supplier-city-follows-part-city>
       <supplier id="S3" name="Blake" status="30" city="Paris"/>
       <part id="P4" name="Screw" color="Red" weight="14" city="London"/>
    </supplier-city-follows-part-city>
    <supplier-city-follows-part-city>
       <supplier id="S3" name="Blake" status="30" city="Paris"/>
       <part id="P6" name="Cog" color="Red" weight="19" city="London"/>
    </supplier-city-follows-part-city>
 </result>
```

7.6 Implementing the W3C XML Query-Use Cases in XSLT

Problem

You need to perform a query operation similar to one of the use cases in *http://www.w3.org/TR/2001/WD-xmlquery-use-cases-20011220*, but you want to use XSLT rather than XQuery (*http://www.w3.org/TR/xquery/*).

Solution

The following examples are XSLT solutions to most of the XML query-use cases presented in the W3C document. The descriptions of each use case are taken almost verbatim from the W3C document.

1. Use case "XMP": experiences and exemplars

   This use case contains several example queries that illustrate requirements gathered by the W3C from the database and document communities. The data use by these queries follows in Examples 7-10 to 7-13.

Example 7-10. bib.xml

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price> 65.95</price>
    </book>

    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
```

Example 7-10. bib.xml (continued)

```
    </book>

    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price> 39.95</price>
    </book>

    <book year="1999">
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor>
               <last>Gerbarg</last><first>Darcy</first>
                <affiliation>CITI</affiliation>
        </editor>
            <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>

</bib>
```

Example 7-11. reviews.xml

```
reviews>
    <entry>
        <title>Data on the Web</title>
        <price>34.95</price>
        <review>
               A very good discussion of semi-structured database
               systems and XML.
        </review>
    </entry>
    <entry>
        <title>Advanced Programming in the Unix environment</title>
        <price>65.95</price>
        <review>
               A clear and detailed discussion of UNIX programming.
        </review>
    </entry>
    <entry>
        <title>TCP/IP Illustrated</title>
        <price>65.95</price>
        <review>
               One of the best books on TCP/IP.
        </review>
    </entry>
</reviews>
```

Example 7-12. books.xml

```
<chapter>
    <title>Data Model</title>
    <section>
        <title>Syntax For Data Model</title>
    </section>
    <section>
        <title>XML</title>
        <section>
            <title>Basic Syntax</title>
        </section>
        <section>
            <title>XML and Semistructured Data</title>
        </section>
    </section>
</chapter>
```

Example 7-13. prices.xml

```
<prices>
    <book>
        <title>Advanced Programming in the Unix environment</title>
        <source>www.amazon.com</source>
        <price>65.95</price>
    </book>
    <book>
        <title>Advanced Programming in the Unix environment </title>
        <source>www.bn.com</source>
        <price>65.95</price>
    </book>
    <book>
        <title> TCP/IP Illustrated </title>
        <source>www.amazon.com</source>
        <price>65.95</price>
    </book>
    <book>
        <title> TCP/IP Illustrated </title>
        <source>www.bn.com</source>
        <price>65.95</price>
    </book>
    <book>
        <title>Data on the Web</title>
        <source>www.amazon.com</source>
        <price>34.95</price>
    </book>
    <book>
        <title>Data on the Web</title>
        <source>www.bn.com</source>
        <price>39.95</price>
    </book>
</prices>
```

Question 1. List books in *bib.xml* published by Addison-Wesley after 1991, including their year and title:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:import href="copy.xslt"/>

<xsl:template match="book[publisher = 'Addison-Wesley' and @year > 1991]">
  <xsl:copy-of select="."/>
</xsl:template>

<xsl:template match="book"/>

</xsl:stylesheet>
```

Question 2. Create a flat list of all the title-author pairs from *bib.xml*, with each pair enclosed in a "result" element:

```
<xsl:template match="/">
<results>
  <xsl:apply-templates select="bib/book/author"/>
</results>
</xsl:template>

<xsl:template match="author">
  <result>
    <xsl:copy-of select="preceding-sibling::title"/>
    <xsl:copy-of select="."/>
  </result>
</xsl:template>
```

Question 3. For each book in *bib.xml*, list the title and authors, grouped inside a "result" element:

```
<xsl:template match="bib">
  <results>
    <xsl:for-each select="book">
    <result>
      <xsl:copy-of select="title"/>
      <xsl:copy-of select="author"/>
    </result>
   </xsl:for-each>
  </results>
</xsl:template>
```

Question 4. For each author in *bib.xml*, list the author's name and the titles of all books by that author, grouped inside a "result" element:

```
<xsl:template match="/">
<results>
  <xsl:for-each select="//author[not(.=preceding::author)]">
    <result>
      <xsl:copy-of select="."/>
      <xsl:for-each select="/bib/book[author=current()]">
        <xsl:copy-of select="title"/>
```

```
      </xsl:for-each>
    </result>
  </xsl:for-each>
  </results>
```

Question 5. For each book found on both *http://www.bn.com* (*bib.xml*) and *http://www.amazon.com* (*reviews.xml*), list the title of the book and its price from each source:

```
<xsl:variable name="bn" select="document('bib.xml')"/>
<xsl:variable name="amazon" select="document('reviews.xml')"/>

<!--Solution 1 -->
<xsl:template match="/">
  <books-with-prices>
  <xsl:for-each select="$bn//book[title = $amazon//entry/title]">
    <book-with-prices>
      <xsl:copy-of select="title"/>
      <price-amazon><xsl:value-of
      select="$amazon//entry[title=current( )/title]/price"/></price-amazon>
      <price-bn><xsl:value-of select="price"/></price-bn>
    </book-with-prices>
  </xsl:for-each>
  </books-with-prices>
</xsl:template>

<!--Solution 2-->
<xsl:template match="/">
  <books-with-prices>
  <xsl:for-each select="$bn//book">
    <xsl:variable name="bn-book" select="."/>
    <xsl:for-each select="$amazon//entry[title=$bn-book/title]">
      <book-with-prices>
        <xsl:copy-of select="title"/>
        <price-amazon><xsl:value-of select="price"/></price-amazon>
        <price-bn><xsl:value-of select="$bn-book/price"/></price-bn>
      </book-with-prices>
    </xsl:for-each>
  </xsl:for-each>
  </books-with-prices>
</xsl:template>
```

Question 6. For each book that has at least one author, list the title and first two authors, as well as an empty "et-al" element if the book has additional authors:

```
<xsl:template match="bib">
  <xsl:copy>
    <xsl:for-each select="book[author]">
      <xsl:copy>
        <xsl:copy-of select="title"/>
        <xsl:copy-of select="author[position( ) &lt;= 2]"/>
        <xsl:if test="author[3]">
        <et-al/>
```

```
        </xsl:if>
      </xsl:copy>
    </xsl:for-each>
  </xsl:copy>
</xsl:template>
```

Question 7. List the titles and years of all books published by Addison-Wesley after 1991, in alphabetic order:

```
<xsl:template match="bib">
  <xsl:copy>
    <xsl:for-each select="book[publisher = 'Addison-Wesley'
          and @year > 1991]">
      <xsl:sort select="title"/>
      <xsl:copy>
        <xsl:copy-of select="@year"/>
        <xsl:copy-of select="title"/>
      </xsl:copy>
    </xsl:for-each>
  </xsl:copy>
</xsl:template>
```

Question 8. In the document *books.xml*, find all section or chapter titles that contain the word "XML", regardless of the nesting level:

```
<xsl:template match="/">
<results>
  <xsl:copy-of select="(//chapter/title |
  //section/title)[contains(.,'XML')]"/>
</results>
</xsl:template>
```

Question 9. In the document *prices.xml*, find the minimum price for each book in the form of a "minprice" element with the book title as its title attribute:

```
<xsl:include href="../math/math.min.xslt"/>

<xsl:template match="/">
<results>
  <xsl:for-each select="//book/title[not(. = ./preceding::title)]">
    <xsl:variable name="min-price">
      <xsl:call-template name="math:min">
        <xsl:with-param name="nodes" select="//book[title =
                                        current( )]/price"/>
      </xsl:call-template>
    </xsl:variable>
    <minprice title="{.}">
      <price><xsl:value-of select="$min-price"/></prices>
    </minprice>
  </xsl:for-each>
</results>
</xsl:template>
```

Question 10. For each book with an author, return the book with its title and authors. For each book with an editor, return a reference with the book title and the editor's affiliation:

```
<xsl:template match="bib">
<xsl:copy>
  <xsl:for-each select="book[author]">
    <xsl:copy>
      <xsl:copy-of select="title"/>
      <xsl:copy-of select="author"/>
    </xsl:copy>
  </xsl:for-each>

  <xsl:for-each select="book[editor]">
    <reference>
      <xsl:copy-of select="title"/>
      <org><xsl:value-of select="editor/affiliation"/></org>
    </reference>
  </xsl:for-each>
  </xsl:copy>
</xsl:template>
```

Question 11. Find pairs of books that have different titles but the same set of authors (possibly in a different order):

```
<xsl:include href="query.equal-values.xslt"/>

<xsl:template match="bib">
  <xsl:copy>
    <xsl:for-each select="book[author]">
      <xsl:variable name="book1" select="."/>
      <xsl:for-each select="./following-sibling::book[author]">
        <xsl:variable name="same-authors">
          <xsl:call-template name="query:equal-values">
            <xsl:with-param name="nodes1" select="$book1/author"/>
            <xsl:with-param name="nodes2" select="author"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:if test="string($same-authors)">
          <book-pair>
            <xsl:copy-of select="$book1/title"/>
            <xsl:copy-of select="title"/>
          </book-pair>
        </xsl:if>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:copy>
</xsl:template>
```

2. Use case "TREE": queries that preserve hierarchy.

   Some XML document types have a very flexible structure in which text is mixed with elements and many elements are optional. These document-types show a wide variation in structure from one document to another. In these types of documents, the ways in which elements are ordered and nested are usually quite important. An XML query language should have the ability to extract elements from documents while preserving their original hierarchy. This use-case illustrates this requirement by means of a flexible document type named Book.

The DTD and XML data used by these queries follows in Examples 7-14 to 7-15.

Example 7-14. book.dtd

```
<!ELEMENT book (title, author+, section+)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT section (title, (p | figure | section)* )>
  <!ATTLIST section
      id         ID    #IMPLIED
      difficulty CDATA #IMPLIED>
  <!ELEMENT p (#PCDATA)>
  <!ELEMENT figure (title, image)>
  <!ATTLIST figure
       width   CDATA   #REQUIRED
       height  CDATA   #REQUIRED >
  <!ELEMENT image EMPTY>
  <!ATTLIST image
       source  CDATA   #REQUIRED >
```

Example 7-15. book.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book SYSTEM "book.dtd">
<book>
  <title>Data on the Web</title>
  <author>Serge Abiteboul</author>
  <author>Peter Buneman</author>
  <author>Dan Suciu</author>
  <section id="intro" difficulty="easy" >
    <title>Introduction</title>
    <p>Text ... </p>
    <section>
      <title>Audience</title>
      <p>Text ... </p>
    </section>
    <section>
      <title>Web Data and the Two Cultures</title>
      <p>Text ... </p>
      <figure height="400" width="400">
        <title>Traditional client/server architecture</title>
        <image source="csarch.gif"/>
      </figure>
      <p>Text ... </p>
    </section>
  </section>
  <section id="syntax" difficulty="medium" >
    <title>A Syntax For Data</title>
    <p>Text ... </p>
    <figure height="200" width="500">
      <title>Graph representations of structures</title>
      <image source="graphs.gif"/>
    </figure>
    <p>Text ... </p>
```

Example 7-15. book.xml (continued)

```
    <section>
      <title>Base Types</title>
      <p>Text ... </p>
    </section>
    <section>
      <title>Representing Relational Databases</title>
      <p>Text ... </p>
      <figure height="250" width="400">
        <title>Examples of Relations</title>
        <image source="relations.gif"/>
      </figure>
    </section>
    <section>
      <title>Representing Object Databases</title>
      <p>Text ... </p>
    </section>
  </section>
</book>
```

Question 1. Prepare a (nested) table of contents for *Book1*, listing all the sections and their titles. Preserve the original attributes of each `<section>` element, if any exist:

```
<xsl:template match="book">
  <toc>
    <xsl:apply-templates/>
  </toc>
</xsl:template>

<!-- Copy element of toc -->
<xsl:template match="section | section/title | section/title/text( )">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
      <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>

<!-- Supress other elements -->
<xsl:template match="* | text( )"/>
```

Question 2. Prepare a (flat) figure list for *Book1*, listing all figures and their titles. Preserve the original attributes of each `<figure>` element, if any exist:

```
<xsl:template match="book">
  <figlist>
    <xsl:for-each select=".//figure">
      <xsl:copy>
        <xsl:copy-of select="@*"/>
        <xsl:copy-of select="title"/>
      </xsl:copy>
    </xsl:for-each>
  </figlist>
</xsl:template>
```

Question 3. How many sections are in *Book1*, and how many figures?

```
<xsl:template match="/">
  <section-count><xsl:value-of select="count(//section)"/></section-count>
  <figure-count><xsl:value-of select="count(//figure)"/></figure-count>
</xsl:template>
```

Question 4. How many top-level sections are in *Book1*?

```
<xsl:template match="book">
  <top_section_count>
     <xsl:value-of select="count(section)"/>
  </top_section_count>
</xsl:template>
```

Question 5. Make a flat list of the section elements in *Book1*. In place of its original attributes, each section element should have two attributes, containing the title of the section and the number of figures immediately contained in the section:

```
<xsl:template match="book">
<section_list>
  <xsl:for-each select=".//section">
    <section title="{title}" figcount="{count(figure)}"/>
  </xsl:for-each>
</section_list>
</xsl:template>
```

Question 6. Make a nested list of the section elements in *Book1*, preserving their original attributes and hierarchy. Inside each section element, include the title of the section and an element that includes the number of figures immediately contained in the section. See Examples 7-16 and 7-17.

Example 7-16. The solution as I would interpret the English requirements

```
<xsl:template match="book">
<toc>
  <xsl:apply-templates select="section"/>
</toc>
</xsl:template>

<xsl:template match="section">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:copy-of select="title"/>
    <figcount><xsl:value-of select="count(figure)"/></figcount>
    <xsl:apply-templates select="section"/>
  </xsl:copy>
</xsl:template>
```

Example 7-17. What the W3C use case wants based on a sample result and XQuery

```
<xsl:template match="book">
<toc>
  <xsl:for-each select="//section">
    <xsl:sort select="count(ancestor::section)"/>
    <xsl:apply-templates select="."/>
```

Example 7-17. What the W3C use case wants based on a sample result and XQuery (continued)

```
  </xsl:for-each>
</toc>
</xsl:template>

<xsl:template match="section">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:copy-of select="title"/>
    <figcount><xsl:value-of select="count(figure)"/></figcount>
    <xsl:apply-templates select="section"/>
  </xsl:copy>
</xsl:template>
```

3. Use case "SEQ": queries based on sequence.

   This use case illustrates queries based on the sequence in which elements appear in a document. Although sequence is not significant in most traditional database systems or object systems, it can be important in structured documents. This use case presents a series of queries based on a medical report:

```
<!DOCTYPE report [
  <!ELEMENT report (section*)>
  <!ELEMENT section (section.title, section.content)>
  <!ELEMENT section.title  (#PCDATA )>
  <!ELEMENT section.content  (#PCDATA | anesthesia | prep
                            | incision | action | observation )*>
  <!ELEMENT anesthesia (#PCDATA)>
  <!ELEMENT prep ( (#PCDATA | action)* )>
  <!ELEMENT incision ( (#PCDATA | geography | instrument)* )>
  <!ELEMENT action ( (#PCDATA | instrument )* )>
  <!ELEMENT observation (#PCDATA)>
  <!ELEMENT geography (#PCDATA)>
  <!ELEMENT instrument (#PCDATA)>
]>
<report>
  <section>
    <section.title>Procedure</section.title>
     <section.content>
      The patient was taken to the operating room where she was placed
      in supine position and
      <anesthesia>induced under general anesthesia.</anesthesia>
      <prep>
        <action>A Foley catheter was placed to decompress the bladder</action>
        and the abdomen was then prepped and draped in sterile fashion.
      </prep>
      <incision>
        A curvilinear incision was made
        <geography>in the midline immediately infraumbilical</geography>
        and the subcutaneous tissue was divided
        <instrument>using electrocautery.</instrument>
      </incision>
      The fascia was identified and
      <action>#2 O Maxon stay sutures were placed on each side of the midline.
      </action>
```

```
      <incision>
        The fascia was divided using
        <instrument>electrocautery</instrument>
        and the peritoneum was entered.
      </incision>
      <observation>The small bowel was identified.</observation>
      and
      <action>
        the
        <instrument>Hasson trocar</instrument>
        was placed under direct visualization.
      </action>
      <action>
        The
        <instrument>trocar</instrument>
        was secured to the fascia using the stay sutures.
      </action>
    </section.content>
  </section>
</report>
```

Question 1. In the Procedure section of *Report1*, what instruments were used in the second incision?

```
<xsl:template match="section[section.title = 'Procedure']">
<xsl:copy-of select="(.//incision)[2]/instrument"/>
</xsl:template>
```

Question 2. In the Procedure section of *Report1*, what are the first two instruments to be used?

```
<xsl:template match="section[section.title = 'Procedure']">
<xsl:copy-of select="(.//instrument)[position( ) &lt;= 2]"/>
</xsl:template>
```

Question 3. In *Report1*, what instruments were used in the first two actions after the second incision?

```
<xsl:template match="report">
<!-- i2 = Second incision in the entire report -->
<xsl:variable name="i2" select="(.//incision)[2]"/>
<!-- Of all the actions following i2
     get the instruments used in the first two -->
<xsl:copy-of
     select="($i2/following::action)[position( ) &lt;= 2]/instrument"/>
</xsl:template>
```

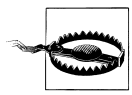
Question 4. In *Report1*, find "Procedure" sections for which no anesthesia element occurs before the first incision:

```
<xsl:template match="section[section.title = 'Procedure']">
  <xsl:variable name="i1" select="(.//incision)[1]"/>
  <xsl:if test=".//anesthesia[preceding::incision = $i1]">
    <xsl:copy-of select="current( )"/>
  </xsl:if>
</xsl:template>
```

If the result is not empty then a major lawsuit is soon to follow!

Question 5. In *Report1*, what happened between the first and second incision?

```
<xsl:template match="report">
<critical_sequence>
  <!-- i1 = First incision in the entire report -->
  <xsl:variable name="i1" select="(.//incision)[1]"/>
  <!-- i2 = Second incision in the entire report -->
  <xsl:variable name="i2" select="(.//incision)[2]"/>
  <!-- copy all sibling nodes following i1
     that don't have a preceding element i2 and are not themeseves i2 -->
  <xsl:for-each select="$i1/following-sibling::node( )
              [not(./preceding::incision = $i2) and not(. = $i2)]">
    <xsl:copy-of select="."/>
  </xsl:for-each>
</critical_sequence>
</xsl:template>
```



In Questions 4 and 5, I assume that the string values of incision elements are unique. This is true in the sample data, but may not be true in the most general case. To be precise, you should apply Recipe 4.2. For example, in Question 4, the test should be:

```
test=".//anesthesia[
count(./preceding::incision | $i1) =
count(./preceding::incision)]"
```

4. Use case "R": access to relational data.

   One important use of an XML query language is the access of data stored in relational databases. This use case describes one possible way in which this access might be accomplished. A relational database system might present a view in which each table (relation) takes the form of an XML document. One way to represent a database table as an XML document is to allow the document element to represent the table itself and each row (tuple) inside the table to be represented by a nested element. Inside the tuple-elements, each column is in turn represented by a nested element. Columns that allow null values are represented by optional elements, and a missing element denotes a null value.

   For example, consider a relational database used by an online auction. The auction maintains a USERS table containing information on registered users, each identified by a unique user ID that can either offer items for sale or bid on items. An ITEMS table lists items currently or recently for sale, with the user ID of the user who offered each item. A BIDS table contains all bids on record, keyed by the user ID of the bidder and the number of the item to which the bid applies.

   Due to the large number of queries in this use case, you will only implement a subset. Implementing the others is a nice exercise if you wish to strengthen your XSLT skills. See Examples 7-18 to 7-20.

Example 7-18. users.xml

```
<users>
  <user_tuple>
    <userid>U01</userid>
    <name>Tom Jones</name>
    <rating>B</rating>
  </user_tuple>
  <user_tuple>
    <userid>U02</userid>
    <name>Mary Doe</name>
    <rating>A</rating>
  </user_tuple>
  <user_tuple>
    <userid>U03</userid>
    <name>Dee Linquent</name>
    <rating>D</rating>
  </user_tuple>
  <user_tuple>
    <userid>U04</userid>
    <name>Roger Smith</name>
    <rating>C</rating>
  </user_tuple>
  <user_tuple>
    <userid>U05</userid>
    <name>Jack Sprat</name>
    <rating>B</rating>
  </user_tuple>
  <user_tuple>
    <userid>U06</userid>
    <name>Rip Van Winkle</name>
    <rating>B</rating>
  </user_tuple>
</users>
```

Example 7-19. items.xml

```
<items>
  <item_tuple>
    <itemno>1001</itemno>
    <description>Red Bicycle</description>
    <offered_by>U01</offered_by>
    <start_date>99-01-05</start_date>
    <end_date>99-01-20</end_date>
    <reserve_price>40</reserve_price>
  </item_tuple>
  <item_tuple>
    <itemno>1002</itemno>
    <description>Motorcycle</description>
    <offered_by>U02</offered_by>
    <start_date>99-02-11</start_date>
    <end_date>99-03-15</end_date>
    <reserve_price>500</reserve_price>
  </item_tuple>
```

Example 7-19. items.xml (continued)

```
  <item_tuple>
    <itemno>1003</itemno>
    <description>Old Bicycle</description>
    <offered_by>U02</offered_by>
    <start_date>99-01-10</start_date>
    <end_date>99-02-20</end_date>
    <reserve_price>25</reserve_price>
  </item_tuple>
  <item_tuple>
    <itemno>1004</itemno>
    <description>Tricycle</description>
    <offered_by>U01</offered_by>
    <start_date>99-02-25</start_date>
    <end_date>99-03-08</end_date>
    <reserve_price>15</reserve_price>
  </item_tuple>
  <item_tuple>
    <itemno>1005</itemno>
    <description>Tennis Racket</description>
    <offered_by>U03</offered_by>
    <start_date>99-03-19</start_date>
    <end_date>99-04-30</end_date>
    <reserve_price>20</reserve_price>
  </item_tuple>
  <item_tuple>
    <itemno>1006</itemno>
    <description>Helicopter</description>
    <offered_by>U03</offered_by>
    <start_date>99-05-05</start_date>
    <end_date>99-05-25</end_date>
    <reserve_price>50000</reserve_price>
  </item_tuple>
  <item_tuple>
    <itemno>1007</itemno>
    <description>Racing Bicycle</description>
    <offered_by>U04</offered_by>
    <start_date>99-01-20</start_date>
    <end_date>99-02-20</end_date>
    <reserve_price>200</reserve_price>
  </item_tuple>
  <item_tuple>
    <itemno>1008</itemno>
    <description>Broken Bicycle</description>
    <offered_by>U01</offered_by>
    <start_date>99-02-05</start_date>
    <end_date>99-03-06</end_date>
    <reserve_price>25</reserve_price>
  </item_tuple>
</items>
```

Example 7-20. bids.xml

```
<bids>
  <bid_tuple>
    <userid>U02</userid>
    <itemno>1001</itemno>
    <bid> 35</bid>
    <bid_date>99-01-07 </bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U04</userid>
    <itemno>1001</itemno>
    <bid>40</bid>
    <bid_date>99-01-08</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U02</userid>
    <itemno>1001 </itemno>
    <bid>45</bid>
    <bid_date>99-01-11</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U04</userid>
    <itemno>1001</itemno>
    <bid>50</bid>
    <bid_date>99-01-13</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U02</userid>
    <itemno>1001</itemno>
    <bid>55</bid>
    <bid_date>99-01-15</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U01</userid>
    <itemno>1002</itemno>
    <bid>400</bid>
    <bid_date>99-02-14</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U02</userid>
    <itemno>1002</itemno>
    <bid>600</bid>
    <bid_date>99-02-16</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U03</userid>
    <itemno>1002</itemno>
    <bid>800</bid>
    <bid_date>99-02-17</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U04</userid>
    <itemno>1002</itemno>
```

Example 7-20. bids.xml (continued)

```
    <bid>1000</bid>
    <bid_date>99-02-25</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U02</userid>
    <itemno>1002</itemno>
    <bid>1200</bid>
    <bid_date>99-03-02</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U04</userid>
    <itemno>1003</itemno>
    <bid>15</bid>
    <bid_date>99-01-22</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U05</userid>
    <itemno>1003</itemno>
    <bid>20</bid>
    <bid_date>99-02-03</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U01</userid>
    <itemno>1004</itemno>
    <bid>40</bid>
    <bid_date>99-03-05</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U03</userid>
    <itemno>1007</itemno>
    <bid>175</bid>
    <bid_date>99-01-25</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U05</userid>
    <itemno>1007</itemno>
    <bid>200</bid>
    <bid_date>99-02-08</bid_date>
  </bid_tuple>
  <bid_tuple>
    <userid>U04</userid>
    <itemno>1007</itemno>
    <bid>225</bid>
    <bid_date>99-02-12</bid_date>
  </bid_tuple>
</bids>
```

Question 1. List the item number and description of all bicycles that currently have an auction in progress, ordered by item number:

```
<xsl:include href="../date/date.date-time.xslt"/>

<!-- To make the result come out like the W3C example -->
<xsl:param name="today" select="'1999-01-21'"/>

<xsl:template match="items">

  <xsl:variable name="today-abs">
    <xsl:call-template name="date:date-to-absolute-day">
      <xsl:with-param name="date" select="$today"/>
    </xsl:call-template>
  </xsl:variable>

<result>
  <xsl:for-each select="item_tuple">
    <xsl:sort select="itemno" data-type="number"/>

    <xsl:variable name="start-abs">
      <xsl:call-template name="date:date-to-absolute-day">
        <xsl:with-param name="date" select="start_date"/>
      </xsl:call-template>
    </xsl:variable>

    <xsl:variable name="end-abs">
      <xsl:call-template name="date:date-to-absolute-day">
        <xsl:with-param name="date" select="end_date"/>
      </xsl:call-template>
    </xsl:variable>

    <xsl:if test="$start-abs &lt;= $today-abs and $end-abs >=
        $today-abs and contains(description, 'Bicycle')">
      <xsl:copy>
        <xsl:copy-of select="itemno"/>
        <xsl:copy-of select="description"/>
      </xsl:copy>
    </xsl:if>

  </xsl:for-each>
</result>
</xsl:template>
```

Question 2. For all bicycles, list the item number, description, and highest bid (if any), ordered by item number:

```
<xsl:include href="../math/math.max.xslt"/>

<xsl:template match="items">

<result>
  <xsl:for-each select="item_tuple[contains(description,'Bicycle')]">
    <xsl:sort select="itemno" data-type="number"/>

  <xsl:variable name="bids"
     select="document('bids.xml')//bid_tuple[itemno=current( )/itemno]/bid"/>
```

```
  <xsl:variable name="high-bid">
    <xsl:call-template name="math:max">
      <xsl:with-param name="nodes" select="$bids"/>
    </xsl:call-template>
  </xsl:variable>

  <xsl:copy>
    <xsl:copy-of select="itemno"/>
    <xsl:copy-of select="description"/>
    <high_bid><xsl:if test="$bids"><xsl:value-of
          select="$high-bid"/></xsl:if></high_bid>
  </xsl:copy>

  </xsl:for-each>
</result>
</xsl:template>
```

Question 3. Find cases when a user with a rating worse (alphabetically, greater) than "C" offers an item with a reserve price of more than 1,000:

```
<!-- Not strictly nec. but spec does not define ratings system so we derive
it dynamically! -->
<xsl:variable name="ratings">
  <xsl:for-each select="document('users.xml')//user_tuple/rating">
    <xsl:sort select="." data-type="text"/>
    <xsl:if test="not(. = ./preceding::rating)">
      <xsl:value-of select="."/>
    </xsl:if>
  </xsl:for-each>
</xsl:variable>

<xsl:template match="items">
<result>
  <xsl:for-each select="item_tuple[reserve_price > 1000]">

  <xsl:variable name="user" select="document('users.xml')//user_tuple[userid
  = current()/offered_by]"/>

  <xsl:if test="string-length(substring-before($ratings,$user/rating)) >
  string-length(substring-before($ratings,'C'))">
    <warning>
      <xsl:copy-of select="$user/name"/>
      <xsl:copy-of select="$user/rating"/>
      <xsl:copy-of select="description"/>
      <xsl:copy-of select="reserve_price"/>
    </warning>
  </xsl:if>
  </xsl:for-each>
</result>
</xsl:template>
```

Question 4. List item numbers and descriptions of items that have no bids:

```
<xsl:template match="items">
<result>
  <xsl:for-each select="item_tuple">

  <xsl:if test="not(document('bids.xml')//bid_tuple[itemno =
  current()/itemno])">
    <no_bid_item>
      <xsl:copy-of select="itemno"/>
      <xsl:copy-of select="description"/>
    </no_bid_item>
  </xsl:if>

  </xsl:for-each>
</result>
</xsl:template>
```

5. Use case "SGML": Standard Generalized Markup Language.

   The example document and queries in this use case were first created for a 1992 conference on Standard Generalized Markup Language (SGML). For your use, the Document Type Definition (DTD) and example document are translated from SGML to XML.

   This chapter does not implement these queries because they are not significantly different from queries in other use cases.

6. Use case "TEXT": full-text search.

   This use case is based on company profiles and a set of news documents that contain data for PR, mergers, and acquisitions. Given a company, the use case illustrates several different queries for searching text in news documents and different ways of providing query results by matching the information from the company profile and news content.

   In this use case, searches for company names are interpreted as word-based. The words in a company name may be in any case and separated by any kind of whitespace.

   All queries can be expressed in XSLT 1.0. However, doing so can result in the need for a lot of text-search machinery. For example, the most difficult queries require a mechanism for testing the existence of any member of a set of text values in another string. Furthermore, many queries require testing of text subunits, such as sentence boundaries.

   Based on techniques covered in Chapter 1, it should be clear that these problems have solutions in XSLT. However, if you will do a lot text querying in XSLT, you will need a generic library of text-search utilities. Developing generic libraries is the focus of Chapter 14, which will revisit some of the most complex full-text queries. For now, you will solve two of the most straightforward text-search problems in the W3C document. This chapter lists the others to give a sense of why these queries can be challenging for XSLT 1.0. The difficult parts are emphasized.

Question 1. Find all news items in which the name "Foobar Corporation" appears in the title:

```
<xsl:template match="news">
<result>
  <xsl:copy-of select="news_item/title[contains(., 'Foobar Corporation')]"/>
</result>
</xsl:template>
```

Question 2. For each news item that is relevant to the Gorilla Corporation, create an "item summary" element. The content of the item summary is the title, date, and first paragraph of the news item, separated by periods. A news item is relevant if the name of the company is mentioned anywhere within the content of the news item:

```
<xsl:template match="news">
<result>
  <xsl:for-each select="news_item[contains(content,'Gorilla Corporation')]">
    <item_summary>
      <xsl:value-of select="normalize-space(title)"/>. <xsl:text/>
      <xsl:value-of select="normalize-space(date)"/>. <xsl:text/>
      <xsl:value-of select="normalize-space(content/par[1])"/>
   </item_summary>
  </xsl:for-each>
</result>
</xsl:template>
```

7. Use case "PARTS": recursive parts explosion

   This use case illustrates how a recursive query might can construct a hierarchical document of arbitrary depth from flat structures stored in a database.

   This use case is based on a "parts explosion" database that contains information about how parts are used in other parts.

   The input to the use case is a "flat" document in which each different part is represented by a `<part>` element with `partid` and name attributes. Each part may or may not be part of a larger part; if so, the `partid` of the larger part is contained in a `partof` attribute. This input document might be derived from a relational database in which each part is represented by a table row with `partid` as primary key and `partof` as a foreign key referencing `partid`.

   The challenge of this use case is to write a query that converts the "flat" representation of the parts explosion, based on foreign keys, into a hierarchical representation in which part containment is represented by the document structure.

   The input data set uses the following DTD:

```
<!DOCTYPE partlist [
    <!ELEMENT partlist (part*)>
    <!ELEMENT part EMPTY>
    <!ATTLIST part
          partid  CDATA  #REQUIRED
          partof  CDATA  #IMPLIED
          name    CDATA  #REQUIRED>
]>
```

Although the partid and partof attributes could have been of type ID and IDREF, respectively, in this schema they are treated as character data, possibly materialized in a straightforward way from a relational database. Each partof attribute matches exactly one partid. Parts having no partof attribute are not contained in any other part.

The output data conforms to the following DTD:

```
<!DOCTYPE parttree [
    <!ELEMENT parttree (part*)>
    <!ELEMENT part (part*)>
    <!ATTLIST part
          partid  CDATA  #REQUIRED
          name    CDATA  #REQUIRED>
]>
```

Sample data conforming to that DTD might look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<partlist>
  <part partid="0" name="car"/>
  <part partid="1" partof="0" name="engine"/>
  <part partid="2" partof="0" name="door"/>
  <part partid="3" partof="1" name="piston"/>
  <part partid="4" partof="2" name="window"/>
  <part partid="5" partof="2" name="lock"/>
  <part partid="10" name="skateboard"/>
  <part partid="11" partof="10" name="board"/>
  <part partid="12" partof="10" name="wheel"/>
  <part partid="20" name="canoe"/>
</partlist>
```

Question 1. Convert the sample document from "partlist" to "parttree" format (see the DTD section for definitions). In the result document, part containment is represented by containment of one <part> element inside another. Each part that is not part of any other part should appear as a separate top-level element in the output document:

```
<xsl:template match="partlist">
  <parttree>
     <!-- Start with the part that is not part of anything -->
    <xsl:apply-templates select="part[not(@partof)]"/>
  </parttree>
</xsl:template>

<xsl:template match="part">
  <part partid="{@partid}" name="{@name}">
    <xsl:apply-templates select="../part[@partof = current( )/@partid]"/>
  </part>
</xsl:template>
```

It turns out that this sort of transformation is easier to code and understand in XSLT than in XQuery. For comparison, here is the XQuery solution offered by the W3C paper:

```
define function one_level (element $p) returns element
{
    <part partid="{ $p/@partid }"
          name="{ $p/@name }" >
        {
            for $s in document("partlist.xml")//part
            where $s/@partof = $p/@partid
            return one_level($s)
        }
    </part>
}

<parttree>
  {
    for $p in document("partlist.xml")//part[empty(@partof)]
    return one_level($p)
  }
</parttree>
```

Even without a detailed understanding of XQuery, you should be able to see that the XQuery solution is needed to explicitly implement the recursion while XSLT's `apply-templates` and pattern matching allow a more declarative solution. Granted, the difference is not that dramatic, but I find XSLT more elegant for this type of problem.

8. Use case "REF": queries based on references.*

   References are an important aspect of XML. This use case describes a database in which references play a significant role and contains several representative queries that exploit these references.

   Suppose that the file *census.xml* contains an element for each person recorded in a recent census. For each person element, the person's name, job, and spouse (if any) are recorded as attributes. The `spouse` attribute is an IDREF-type attribute that matches the spouse element's ID-type `name` attribute.

   The parent-child relationship among persons is recorded by containment in the element hierarchy. In other words, the element that represents a child is contained within the element that represents the child's father or mother. Due to deaths, divorces, and remarriages, a child might be recorded under either its father or mother (but not both). In this exercise, the term "children of X" includes "children of the spouse of X." For example, if Joe and Martha are spouses, Joe's element contains an element Sam, and Martha's element contains an element Dave, then both Joe's and Martha's children are considered to be Sam and Daveve. Each person in the census has zero, one, or two parents.

* These use cases were dropped from the latest version of the W3C document.

This use case is based on an input document named *census.xml*, with the following DTD:

```
<!DOCTYPE census [
  <!ELEMENT census (person*)>
  <!ELEMENT person (person*)>
  <!ATTLIST person
        name    ID      #REQUIRED
        spouse  IDREF   #IMPLIED
        job     CDATA   #IMPLIED >
]>
```

The following census data describes two friendly families that have several intermarriages:

```
<census>
  <person name="Bill" job="Teacher">
    <person name="Joe" job="Painter" spouse="Martha">
      <person name="Sam" job="Nurse">
        <person name="Fred" job="Senator" spouse="Jane">
        </person>
      </person>
      <person name="Karen" job="Doctor" spouse="Steve">
      </person>
    </person>
    <person name="Mary" job="Pilot">
      <person name="Susan" job="Pilot" spouse="Dave">
      </person>
    </person>
  </person>
  <person name="Frank" job="Writer">
    <person name="Martha" job="Programmer" spouse="Joe">
      <person name="Dave" job="Athlete" spouse="Susan">
      </person>
    </person>
    <person name="John" job="Artist">
      <person name="Helen" job="Athlete">
      </person>
      <person name="Steve" job="Accountant" spouse="Karen">
        <person name="Jane" job="Doctor" spouse="Fred">
        </person>
      </person>
    </person>
  </person>
</census>
```

Question 1. Find Martha's spouse:

```
<xsl:strip-space elements="*"/>

<xsl:template match="person[@spouse='Martha']">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
  </xsl:copy>
</xsl:template>
```

Question 2. Find parents of athletes:

```
<xsl:template match="census">
  <xsl:variable name="everyone"  select="//person"/>
  <result>
     <!-- For each person with children -->
    <xsl:for-each select="$everyone[person]">
      <xsl:variable name="spouse"
          select="$everyone[@spouse=current( )/@name]"/>
      <xsl:if test="./person/@job = 'Athlete' or
                 $spouse/person/@job = 'Athlete'">
        <xsl:copy>
          <xsl:copy-of select="@*"/>
        </xsl:copy>
      </xsl:if>
    </xsl:for-each>
  </result>
</xsl:template>
```

Question 3. Find people who have the same job as one of their parents.

Try it yourself.

Question 4. List names of parents and children who have the same job, and list their jobs:

```
<xsl:template match="census">
  <xsl:variable name="everyone"  select="//person"/>
  <result>
     <!-- For each person with children -->
    <xsl:for-each select="$everyone[person]">

      <xsl:variable name="spouse"
          select="$everyone[@spouse=current( )/@name]"/>

      <xsl:apply-templates select="person[@job = current( )/@job]">
        <xsl:with-param name="parent" select="@name"/>
      </xsl:apply-templates>

      <xsl:apply-templates select="person[@job = $spouse/@job]">
        <xsl:with-param name="parent" select="$spouse/@name"/>
      </xsl:apply-templates>

    </xsl:for-each>
  </result>
</xsl:template>

<xsl:template match="person">
  <xsl:param name="parent"/>
  <match parent="{$parent}"  child="{@name}" job="{@job}"/>
</xsl:template>
```

Question 5. List name-pairs of grandparents and grandchildren:

```
<xsl:template match="census">
  <xsl:variable name="everyone"  select="//person"/>
  <result>
    <!-- For each grandchild -->
```

```
    <xsl:for-each select="$everyone[../../../person]">
        <!-- Get the grandparent1 (guaranteed to exist by for each -->
        <grandparent name="{../../@name}" grandchild="{@name}"/>
        <!-- Get the grandparent2 is grandparent1's  spouse if listed -->
        <xsl:if test="../../@spouse">
          <grandparent name="{../../@spouse}" grandchild="{@name}"/>
        </xsl:if>
       <!-- Get the names of this person's parent's spouse
          (i.e. their mother or father as the case may be) -->
        <xsl:variable name="spouse-of-parent" select="../@spouse"/>
        <!-- Get parents of spouse-of-parent, if present -->
        <xsl:variable name="gp3"
          select="$everyone[person/@name=$spouse-of-parent]"/>
        <xsl:if test="$gp3">
          <grandparent name="{$gp3/@name}" grandchild="{@name}"/>
          <xsl:if test="$gp3/@spouse">
            <grandparent name="{$gp3/@spouse}" grandchild="{@name}"/>
          </xsl:if>
        </xsl:if>
    </xsl:for-each>
  </result>
</xsl:template>
```

Question 6. Find people with no children:

```
<xsl:strip-space elements="*"/>
<xsl:template match="census">
  <xsl:variable name="everyone"  select="//person"/>
  <result>
    <xsl:for-each select="$everyone[not(./person)]">
      <xsl:variable name="spouse"
      select="$everyone[@name = current( )/@spouse]"/>
      <xsl:if test="not ($spouse) or not($spouse/person)">
        <xsl:copy-of select="."/>
      </xsl:if>
    </xsl:for-each>
  </result>
</xsl:template>
```

Question 7. List the names of all Joe's descendants. Show each descendant as an element with the descendant's name as content and his or her marital status and number of children as attributes. Sort the descendants in descending order by number of children and secondarily in alphabetical order by name:

```
<xsl:variable name="everyone" select="//person"/>

<xsl:template match="census">
  <result>
    <xsl:apply-templates select="//person[@name='Joe']"/>
  </result>
</xsl:template>

<xsl:template match="person">
```

```
  <xsl:variable name="all-desc">
    <xsl:call-template name="descendants">
      <xsl:with-param name="nodes" select="."/>
    </xsl:call-template>
  </xsl:variable>

  <xsl:for-each select="exsl:node-set($all-desc)/*">
    <xsl:sort select="count(./* | $everyone[@name = current( )/@spouse]/*)"
    order="descending" data-type="number"/>
    <xsl:sort select="@name"/>
    <xsl:variable name="mstatus"
          select="normalize-space(
               substring('No Yes',boolean(@spouse)* 3+1,3))"/>
    <person married="{$mstatus}"
          nkids="{count(./* | $everyone[@name = current( )/@spouse]/*)}">
          <xsl:value-of select="@name"/>
     </person>
  </xsl:for-each>
</xsl:template>

<xsl:template name="descendants">
  <xsl:param name="nodes"/>
  <xsl:param name="descendants" select="/.."/>

  <xsl:choose>
    <xsl:when test="not($nodes)">
      <xsl:copy-of select="$descendants"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:call-template name="descendants">
        <xsl:with-param name="nodes" select="$nodes[position( ) > 1] |
          $nodes[1]/person | id($nodes[1]/@spouse)/person"/>
        <xsl:with-param name="descendants" select="$descendants |
          $nodes[1]/person | id($nodes[1]/@spouse)/person"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>

</xsl:template>
```

This example accomplishes the query, but it isn't pretty! The complications come from the need to collect all descendants into a node set so they can be sorted. This forces the use of the node-set extension function. It also means that the `id( )` function will not help find the spouse because it only works relative to the node's document. However, the nodes are copies of the original nodes and thus do not have the same document. This situation forces you to go after the spouse elements in a much more cumbersome way by searching for a variable containing all person elements. Contrast this solution to the following XQuery solution:

```
define function descrip (element $e) returns element
{
    let $kids := $e/* union $e/@spouse=>person/*
    let $mstatus :=  if ($e[@spouse]) then "Yes" else "No"
    return
        <person married={ $mstatus } nkids={ count($kids) }>{ $e/@name/text(
) }</person>
}

define function descendants (element $e)
{
    if (empty($e/* union $e/@spouse=>person/*))
    then $e
    else $e union descendants($e/* union $e/@spouse=>person/*)
}

descrip(descendants(//person[@name = "Joe"])) sortby(@nkids descending, .)
```

Discussion

Unlike most other examples in this book, this one is a smorgasbord of prepared meals. Querying XML can mean so many things that are difficult to come up with. The W3C did a decent job classifying the kinds of queries that come up in various domains. The demonstration of these query solutions in XSLT should provide a sound base for approaching many types of query problems.

Due to space considerations, this chapter did not include the XQuery solutions to the previous problems. Nevertheless, contrasting the two approaches is instructive, so I encourage the reader to examine the W3C Query Use Case document.

Providing individual commentary on each query implemented earlier would be impractical. However, most readers with basic XSLT skills should have little trouble deciphering the solution. Many solutions shown have alternate solutions in XSLT. Some of the alternatives may actually be better than the ones in this chapter. My solutions were heavily influenced by the XQuery solution presented in the original W3C document. However, I also tried to vary the XSLT constructs used, sometimes favoring an iterative style (`xsl:for-each`) and other times using the declarative style provided by patterns and `xsl:apply-templates`.