Fortunately, XSLT is not the last word in XML querying, and a number of other XML query languages have been proposed.

18.4.3 XQuery: A Full-Featured Query Language for XML

XPath and XSLT provide certain query facilities for XML documents. However, we have seen that XPath was designed to be lightweight and so can express only simple queries. XSLT has much greater expressive power, but was not designed as a query language; as a result, it has difficulty formulating complex queries.

Among the many languages specifically designed for querying XML, XQL and XML-QL deserve special mention. XQL [Robie et al. 1998] is an extension of XPath; XML-QL [Deutsch et al. 1998, Florescu et al. 1999] is an SQL-style query language, which also builds on ideas borrowed from languages such as OQL (Chapter 17) and Lorel [Abiteboul et al. 1997]. More recently, the best features of XQL and XML-QL were combined in a language called **XQuery**, which has become an official W3C query language for XML [XQuery 2002]. Like XSLT, XQuery uses XPath as a syntax for its path expressions. However, XQuery is generally more succinct and transparent than XSLT when it comes to querying. This section introduces the main features of XQuery.

Selections and joins. Unlike XSLT, XQuery does not use the verbose syntax of XML, as there is no good reason for any query language to do so.[20] Instead, XQuery statements have some similarity with SQL:

FOR	*variable declarations*
WHERE	*condition*
RETURN	*result*

The FOR clause plays the same role as the FROM clause in SQL, and the WHERE clause is borrowed from SQL with the same functionality. The RETURN clause is analogous to SELECT: In SQL, it defines the template for the result relation; in XQuery, it specifies the template for the result document.

At a deeper level, XQuery is strongly influenced by OQL, the object-oriented query language for ODMG databases (Chapter 17). This connection will be apparent from the examples. We start with simple queries against the document shown in Figure 18.19, on page 664, which resides at the URL `http://xyz.edu/transcripts.xml`.

The first query retrieves all students who have ever taken `MAT123`.

```
FOR $t IN document("http://xyz.edu/transcripts.xml")//Transcript
WHERE $t/CrsTaken/@CrsCode = "MAT123"
RETURN $t/Student
```

The FOR clause declares a variable, `$t`, and its range—a set of document nodes. This set is specified using the XPath expression `//Transcript`, which is applied

[20] At least, not for human consumption. XML-based syntax for XQuery is under development, however. For one thing, it can simplify the job of building parsers for XQuery.

to the document obtained using the function `document()`. This function, borrowed from XSLT, returns the root of the document specified by the URL. Hence `$t` ranges over all `Transcript` nodes in the document. To navigate within document trees, XQuery relies on XPath expressions, which are extended with variables. Thus, `$t/CrsTaken/@CrsCode` and `$t/Student` are extended XPath expressions. When `$t` is bound to a `Transcript` node in the document tree, the first expression returns the nodes corresponding to the `CrsCode` attribute of the `CrsTaken` *e*-children of that node. The second expression returns `Student` *e*-children of the node.

The condition `$t/CrsTaken/@CrsCode = "MAT123"` in the WHERE clause has a subtlety that illustrates one very important aspect of the XQuery semantics. Note that `$t/CrsTaken/@CrsCode` is a *set-valued* expression that contains all codes of the courses listed in a transcript, `$t`. On the other hand, `"MAT123"` is a *single* constant. What does it mean to equate a set and a constant? More generally, what does it mean to compare two set-valued XPath expressions, `pathexpr1 op pathexpr2`? In XQuery, such a condition is true if and only if there are elements `e1`$\in$`pathexpr1` and `e2`$\in$`pathexpr2` such that `e1 op e2` is true. Thus, the WHERE condition in the above query selects a subset of the `Transcript` nodes, where each node has at least one `CrsTaken` element with `MAT123` as the value of the `CrsCode` attribute.

The variable `$t` is successively bound to each `Transcript` node that satisfies the WHERE clause and the RETURN clause is executed for each such binding of `$t`. Each execution of the RETURN clause outputs a fragment of the result document, which in our case is the `Student` element contained within the `Transcript` node that constitutes the current value of `$t`. In the example, the following is output:

```
<Student StudId="111111111" Name="John Doe"/>
<Student StudId="123454321" Name="Joe Blow"/>
```

One problem with this output is that it is not a well-formed XML document. It produces a list of `Student` elements, which is not contained within a single parent element. This problem is easy to fix by embedding the above query between a pair of tags.

```
<StudentList>
{
     FOR $t IN document("http://xyz.edu/transcripts.xml")
                                //Transcript
     WHERE $t/CrsTaken/@CrsCode = "MAT123"
     RETURN $t/Student
}
</StudentList>
```

The result is that the FOR clause outputs `Student` elements one by one and places them as children of `StudentList`. In this context, the `StudentList` tag pair is called **element constructor**. The curly braces indicate that the text inside is an expression that is to be evaluated to become the contents of the element denoted by the element constructor. In the following examples, we will omit the outermost element constructor.

Figure 18.19 Transcripts at `http://xyz.edu/transcripts.xml`.

```
<?xml version="1.0" ?>
<Transcripts>
     <Transcript>
          <Student StudId="111111111" Name="John Doe"/>
          <CrsTaken CrsCode="CS308" Semester="F1997" Grade="B"/>
          <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="B"/>
          <CrsTaken CrsCode="EE101" Semester"F1997" Grade="A"/>
          <CrsTaken CrsCode="CS305" Semester="F1995" Grade="A"/>
     </Transcript>
     <Transcript>
          <Student StudId="987654321" Name="Bart Simpson"/>
          <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
          <CrsTaken CrsCode="CS308" Semester="F1994" Grade="B"/>
     </Transcript>
     <Transcript>
          <Student StudId="123454321" Name="Joe Blow"/>
          <CrsTaken CrsCode="CS315" Semester="S1997" Grade="A"/>
          <CrsTaken CrsCode="CS305" Semester="S1996" Grade="A"/>
          <CrsTaken CrsCode="MAT123" Semester="S1996" Grade="C"/>
     </Transcript>
     <Transcript>
          <Student StudId="023456789" Name="Homer Simpson"/>
          <CrsTaken CrsCode="EE101" Semester="F1995" Grade="B"/>
          <CrsTaken CrsCode="CS305" Semester="S1996" Grade="A"/>
     </Transcript>
</Transcripts>
```

The previous example has shown the convenience of using set-valued XPath expressions in the WHERE clause. Before proceeding to more complex queries, a word of caution is in order: set-valued comparisons can let subtle mistakes creep in. One frequent problem is illustrated in the following example. Suppose in the previous example we wanted only those students who took `MAT123` in a particular semester as, for instance, in

`$t/CrsTaken/@CrsCode = "MAT123"` and `$t/CrsTaken/@Semester = "F2002"`

While this condition seems natural, it is incorrect. The correct one is

`$t/CrsTaken[@CrsCode = "MAT123"` and `@Semester = "F2002"]`

The problem with the former expression is that it selects those transcripts which contain a record for `MAT123` and a record that refers to a course for fall 2002. The two records do not need to be the same, however. The second expression avoids this ambiguity.

Figure 18.20 Construction of class rosters from transcripts: first try.

```
FOR $c IN distinct-values(document("http://xyz.edu/transcripts.xml")
                                  //CrsTaken)
RETURN <ClassRoster CrsCode={$c/@CrsCode} Semester={$c/@Semester}>
    {
        FOR $t IN document("http://xyz.edu/transcripts.xml")
                                  //Transcript
        WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and
                          @Semester = $c/@Semester]
        RETURN
                $t/Student
                ORDER BY $t/Student/@StudId
    }
</ClassRoster>
ORDER BY $c/@CrsCode
```

The next example illustrates the restructuring capabilities of XQuery. The `Transcripts` document in Figure 18.19 groups course records around the students who took them. However, the user reading this document might want to reconstruct class lists for each course. In other words, for each course offering in a particular semester, she might want to obtain the list of students who took that course. In XQuery, this can be done directly from the `transcripts.xml` document, as shown in Figure 18.20. In that query, the variable `$c` ranges over the set of all distinct `CrsTaken` nodes in the document `transcripts.xml`. For each such node, a fragment of the result document is constructed as described in the `RETURN` clause.

The query in Figure 18.20 is almost correct, but it has a flaw that we will explain after discussing the new features it exhibits.

First, observe that the query is nested. The interesting point, however, is that the nesting occurs in the `RETURN` clause, which corresponds to the `SELECT` clause in SQL. The SQL specification bans nested queries in the `SELECT` clause, because they do not make sense in the relational model.[21] Indeed, a nested query typically returns a set of tuples whereas the target list of a `SELECT` clause in SQL is a template for a single tuple. In contrast, nested queries in the `RETURN` clause make perfect sense for an XML query language, because the purpose here is to construct a document that can have an arbitrarily complex nested structure. In our case, nesting serves the purpose of embedding student lists into class rosters. We have already encountered the use of nested queries in the target list of a query: This is allowed in the object-oriented query language OQL (see query (17.11) on page 562). Nesting is used there for the same reason it is used in XQuery.

[21] More precisely, SQL permits only those queries that return a single scalar value, such as an integer or a string.

Let us look more closely at the RETURN clause in the query in Figure 18.20, which is executed for each value of the variable $c. First, it constructs the ClassRoster element with the appropriate values for the attributes CrsCode and Semester. Then the nested subquery is invoked to construct the *sorted* list of students who have taken this class (i.e., the given course in the given semester). In it, $t ranges over Transcript elements and the WHERE clause selects those that have a CrsTaken element that matches the semester and course specified in $c. The output consists of elements such as

```
<ClassRoster CrsCode="CS305" Semester="F1995">
     <Student StudId="111111111" Name="John Doe"/>
     <Student StudId="987654321" Name="Bart Simpson"/>
</ClassRoster>
```

which are themselves sorted by the CrsCode attribute that occurs in each roster.

So far so good, except for one thing: John Doe and Bart Simpson received different grades for CS305 in fall 1995, so the FOR clause binds $c to two different CrsTaken elements for that class:

```
<CrsTaken CrsCode="CS305" Semester="F1995" Grade="A"/>
<CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
```

This means that the above ClassRoster element will be output twice. In general, each roster will be output once for every distinct grade received in the corresponding class. One way to overcome this problem is to create a new document that contains a list of all classes and then bind $c to the elements of that list. This can be easily done by selecting all CrsTaken elements from the transcripts document and then stripping off the Grade attribute from each new one.

We will revisit this idea later, but for now we avoid this problem by assuming that the document in Figure 18.21, which resides at the URL http://xyz.edu/classes.xml, already exists. The following query is a reformulation of the faulty query of Figure 18.20 with slight enhancements intended to illustrate the join operation in XQuery:

```
FOR $c IN document("http://xyz.edu/classes.xml")//Class
RETURN
     <ClassRoster CrsCode={$c/@CrsCode} Semester={$c/@Semester}>
          {
               $c/CrsName,
               $c/Instructor,
               FOR $t IN document("http://xyz.edu/transcripts.xml")
                                    //Transcript
               WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and
                                 @Semester = $c/@Semester]
               RETURN
               $t/Student
               ORDER BY $t/Student/@StudId
```

Figure 18.21 Classes at `http://xyz.edu/classes.xml`.

```
<?xml version="1.0" ?>
<Classes>
     <Class CrsCode="CS308" Semester="F1997">
          <CrsName>Software Engineering</CrsName>
          <Instructor>Adrian Jones</Instructor>
     </Class>
     <Class CrsCode="EE101" Semester="F1995">
          <CrsName>Electronic Circuits</CrsName>
          <Instructor>David Jones</Instructor>
     </Class>
     <Class CrsCode="CS305" Semester="F1995">
          <CrsName>Database Systems</CrsName>
          <Instructor>Mary Doe</Instructor>
     </Class>
     <Class CrsCode="CS315" Semester="S1997">
          <CrsName>Transaction Processing</CrsName>
          <Instructor>John Smyth</Instructor>
     </Class>
     <Class CrsCode="MAT123" Semester="F1997">
          <CrsName>Algebra</CrsName>
          <Instructor>Ann White</Instructor>
     </Class>
</Classes>
```

```
          }
     </ClassRoster>
     ORDER BY $c/@CrsCode
```

The change in this new query is that the variable `$c` ranges over the set of all `Class` nodes of the document in Figure 18.21. Since classes do not occur multiple times there, we no longer have the problem of outputting multiple instances of the same roster (and we do not even need to apply the function `distinct-values()`). The result of the new query is enriched by including the course name (`$c/CrsName`) and the instructor (`$c/Instructor`) as child elements in each roster. They are conjoined to the FOR-expression using the **concatenation operator** ",". Thus, this query computes a kind of *equijoin* on the attributes `CrsCode` and `Semester` between the `Transcript` elements of the document in Figure 18.19 and the `Class` elements of the document in Figure 18.21. As remarked at the end of Section 18.4.2, it is difficult to formulate this kind of transformation in XSLT.

Note that in the above query, even if some class has no students, the corresponding element `ClassRoster` will still appear in the result, albeit with empty contents. This is different from the join operation in relational databases, where a tuple in

a CLASS relation with no matching tuples in the TRANSCRIPT relation will not be considered in an equijoin on the attributes `CrsCode` and `Semester`. The kind of join produced by the above XQuery example is known in relational databases as an **outer join** (defined in exercise 5.9, Chapter 5). Nevertheless, it is easy to reformulate the query and make it perform a "real" join by adding an appropriate WHERE clause to the outermost FOR statement:

```
FOR $c IN document("http://xyz.edu/classes.xml")//Class
WHERE document("http://xyz.edu/transcripts.xml")
                      //CrsTaken[@CrsCode = $c/@CrsCode
                                  and @Semester = $c/@Semester]
RETURN
     <ClassRoster CrsCode={$c/@CrsCode} Semester={$c/@Semester}>
          {
               $c/CrsName,
               $c/Instructor,
               FOR $t IN document("http://xyz.edu/transcripts.xml")
                                        //Transcript
               WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and
                                 @Semester = $c/@Semester]
               RETURN
                      $t/Student
                      ORDER BY $t/Student/@StudId
          }
     </ClassRoster>
ORDER BY $c/@CrsCode
```

The purpose of the new WHERE clause is to test whether the document at `http://xyz.edu/transcripts.xml` has `CrsTaken` elements that match the attributes `CrsCode` and `Semester` for the current value of the variable `$c`. This test is performed by the path expression `//CrsTaken[....]` in the first WHERE clause. Only if such elements exist is the corresponding `ClassRoster` element output. Thus, rosters for the classes with no students will not appear in the result.

We will add one final touch to this example before turning our attention to other features of XQuery. Note that in the node set `document("http://xyz.edu/transcripts.xml")` occurs twice in the above query and it is clear that, in general, a same complex expression that denotes a set of nodes can occur multiple times in a query. It is therefore natural to introduce variables that can take node sets as values. This is what the `LET` construct lets you do. The rewritten query is shown in Figure 18.22. In this figure the use of `LET` is still just a syntactic sugar, but soon we will see more substantial uses of this construct.

The semantics of XQuery. So far, we have discussed the various examples informally, without explaining how the actual query evaluation mechanism works. We are now going to clarify these issues.

The FOR clause has the following functions:

Figure 18.22 Construction of class rosters: correct version.

```
LET $trs := document("http://xyz.edu/transcripts.xml")
FOR $c IN document("http://xyz.edu/classes.xml")//Class
WHERE $trs//CrsTaken[@CrsCode = $c/@CrsCode and
                     @Semester = $c/@Semester]
RETURN
     <ClassRoster CrsCode={$c/@CrsCode} Semester={$c/@Semester}>
          {
               $c/CrsName,
               $c/Instructor,
               FOR $t IN $trs//Transcript
               WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and
                                 @Semester = $c/@Semester]
               RETURN
                      $t/Student
                      ORDER BY $t/Student/@StudId
          }
     </ClassRoster>
ORDER BY $c/@CrsCode
```

- To specify the documents to be used in the query.
- To declare variables.
- To bind each variable to its range, which is an *ordered* set of document nodes specified by an XQuery expression. Typically, this is an XPath expression, but, as we shall see later, it can also be a query or a function that returns a list of nodes.

The bindings produced by the FOR clause translate into an ordered list of tuples, each containing a concrete binding for every variable mentioned in the FOR clause. For instance, if the FOR clause declares the variables `$a` and `$b` and binds them to the document nodes `{v,w}` and `{x,y,z}`, respectively, then the following ordered list of tuples will be produced: `{v,x}`, `{v,y}`, `{v,z}`, `{w,x}`, `{w,y}`, `{w,z}`. Each tuple, such as, `{w,x}`, provides a concrete binding—`$a/w`, `$b/x`—to our variables.

Next, the tuples of bindings are filtered through the WHERE condition. Thus, if the condition is `$a/CrsTaken/@CrsCode = $b/Class/@CrsCode` and the document nodes `w` and `x` are such that `w/CrsTaken/@CrsCode = x/Class/@CrsCode`, then the tuple `{w,x}` of bindings for `$a` and `$b` is retained; otherwise, it is discarded. The effect of the WHERE clause is thus a selection of an ordered sublist from the original list of tuples.

Finally, for each surviving tuple of bindings the RETURN expression is instantiated. The result is the creation of a fragment for the output document. The process repeats until all eligible tuple bindings are exhausted.

User-defined functions. XQuery provides a large number of built-in functions, which include all of the core functions available in XPath. It also provides some other useful functions, such as `distinct-values()` and `document()`, and others that we will see later.

More interestingly, a query in XQuery can define a number of functions, which can then be called from within the main FOR-WHERE-RETURN query. Functions can call themselves recursively; they can take singleton nodes as well as collections of nodes as arguments; and they can return primitive types, document nodes, or collections of any of these types. The body of a function is an XQuery expression, which can be general (even queries are treated as expressions). An expression can evaluate to an integer, an element, a list of elements, and so on, and the function returns its result.

Here is an example of a function that counts the number of descendant element nodes in a document fragment rooted at node `$e`:

```
DEFINE FUNCTION countNodes($e AS element) AS integer {
    RETURN
        IF empty($e/*) THEN 0
        ELSE sum(FOR $n IN $e/* RETURN countNodes($n))
            + count($e/*)
}
```

This function definition illustrates a great number of features.

- The declaration `$e AS element` says that the argument to the function must be an element. In particular, it cannot be an attribute or a text node, and it cannot be an integer. The function is said to return an integer. We explain a bit later where such types come from.

  The body of an XQuery function is an **XQuery expression**, which evaluates to a value (an integer, a string, a document node, a list of nodes, etc.). The value of that expression is returned. XQuery expressions are explained next.

- The statement IF-THEN-ELSE is a conditional XQuery expression. The part between IF and THEN must be a Boolean expression—in our case `empty($e/*)`, which uses the built-in function `empty()` to check whether the path expression `$e/*` returns the empty set of document nodes.[22]

  The THEN and ELSE parts must be XQuery expressions. We do not define the full syntax of XQuery expressions here.[23] Typically they are path expressions (which return a set of document nodes), function calls (which can return a primitive type such as `integer` or `string`, a document node, or a set of nodes in the case of user-defined functions), arithmetic expressions, IF-THEN-ELSE conditionals, or full-blown queries (which can return a single document or a list of document fragments).

[22] Recall that the wildcard `*` selects all *e*-children of the current node, so `$e/*` returns the set of all elements that are children of the node assigned to `$e`.

[23] The interested reader is referred to [XQuery 2002].

Figure 18.23 Class rosters constructed with user-defined functions.

```
DEFINE FUNCTION extractClasses($e AS element) AS element* {
  FOR $ct IN $e//CrsTaken
  RETURN <Class CrsCode=$ct/@CrsCode Semester=$ct/@Semester/>
}

<Rosters>
{
  LET $trs := document("http://xyz.edu/transcripts.xml")
  FOR $c IN distinct-values(FOR $d IN $trs RETURN extractClasses($d))
  RETURN
    <ClassRoster CrsCode={$c/@CrsCode} Semester={$c/@Semester}>
      {
        FOR $t IN  $trs//Transcript[CrsTaken/@CrsCode=$c/@CrsCode and
                                    CrsTaken/@Semester=$c/@Semester]
        RETURN $t/Student
        ORDER BY $t/Student/@StudId
      }
    </ClassRoster>
}
</Rosters>
```

In our case, the THEN expression is just a constant and the ELSE part is an arithmetic expression that makes calls to the aggregate functions `sum()` and `count()`. The function `sum()` applies to a collection of numeric values computed by recursive calls to `countNodes()`, and sums them up. The function `count()` counts the number of nodes returned by the path expression `$e/*`.

In the next example, we come back to the "nearly correct" query in Figure 18.20. Recall that the problem was that a class roster might appear multiple times in the result if at least two students in that class receive different grades. One solution that we came up with was to join transcripts with the document in Figure 18.21. XQuery functions make it possible to create an intermediate document similar to that of Figure 18.21 and join it with `Transcripts` in the same query, without relying on the existence of another, external document. The solution is shown in Figure 18.23.

The first part of the query defines the function `extractClasses()`. This function accepts a single element and returns a list of elements. The return type is specified as `element*` using the already familiar keyword `element` and the sequence indicator "*". The result of the function is obtained by evaluating a simple query, which iterates through all `CrsTaken` descendants of the function argument, strips off the `Grade` attribute, and outputs the result as a list of `Class` elements.

The main query appears below the function definition. It returns a document with the top-level tag `Rosters`, which contains a list of individual elements tagged with `ClassRoster`. This query is almost identical to the one in Figure 18.20

except that the variable `$c` ranges over the result produced by the function `extractClasses()`. Also, the WHERE clause has been eliminated and replaced with a selection condition on the XPath expression that binds the variable `$t`. (Selection conditions in XPath expressions were introduced in Section 18.4.1.)

The last example of user-defined functions illustrates how XQuery can perform document transformations of the kind discussed in connection with XSLT. Specifically, we rewrite the XSLT stylesheet in Figure 18.18 on page 661, which traverses an XML document and replaces attributes with elements that have the same name and content. The XQuery equivalent of that program is as follows:

```
DEFINE FUNCTION convertAttribute($a AS attribute) AS element {
  RETURN
    element {name($a)} {data($a)}
}
DEFINE FUNCTION convertElement($e AS attribute) AS element{
  RETURN
    element {name($e)}
    {
      {FOR $a IN $e/@* RETURN convertAttribute($a)},
      IF empty($e/*) THEN $e/text()
      ELSE {FOR $n IN $e/* RETURN convertElement($n)}
    }
}

RETURN convertElement(document("....")/*)
```

The actual query consists of a single call to a previously defined function, `convertElement()`, which takes as an argument a single element node. In this case, the argument is the child node of the document root.[24] Note that, since a well-formed XML document has exactly one topmost element, there is no need for an iteration construct, such as the FOR clause.

The first function, `convertAttribute()`, takes an attribute node as an argument and converts it into the element that has the same name (obtained via a call to the XPath function `name()`). The value of the attribute (obtained via the XQuery function `data()`) becomes the content of the element. The actual element is constructed using a **computed element constructor**, which is specified as `element {`*element name expression*`} {`*contents expression*`}`.

The second function, `convertElement()`, does the bulk of the work. It is called to convert an element node into an attribute-less element with the same name. For a given element, `convertElement()` outputs an element with the same name (again, using a computed element and a call to `name()`), then converts the element's attributes, and finally its *et*-children. To convert attributes, it uses a

[24] We ignore the possibility that there might be other children of the root, such as comments and processing instructions.

FOR-expression to repeatedly invoke `convertAttribute()` on each attribute of the element.[25]

Having finished with attributes, `convertElement()` turns to text nodes and elements. If the element has no *e*-children (as determined by `empty()`),[26] the only child must be a text node (or nothing at all if the element is empty). In the first case, the function emits the text node if it exists; in the second case, it transforms the *e*-children of the current node by calling itself recursively. Again, a list of elements is passed to a function that takes only a single element as a parameter, so this call to `convertElement()` outputs a list of transformed elements.

The above query does not perform quite the same transformation as the XSLT stylesheet in Figure 18.18, on page 661. For instance, if an element has mixed content—its children include elements as well as text nodes[27]—the query ignores the text (because the XPath expression `$e/*` selects only *e*-children) and proceeds to convert the elements. We tackle this problem in the next example, after explaining the relationship between XQuery, XML schemas, and namespaces.

XQuery and data types. The previous queries showed the use of primitive types, such as `integer`, and the XQuery generic types like `element`, and `attribute`. However, XQuery goes much further by integrating smoothly with the XML Schema specification and by allowing the importation and use of the types defined in various XML schemas. In fact, the primitive type `integer` used earlier is not a native XQuery type but rather the one defined by the XML Schema specification, so it should be used in conjunction with the corresponding namespace.

We will now illustrate the integration of XQuery with XML schemas and namespaces using a true XQuery equivalent of the XSLT stylesheet in Figure 18.18 on page 661. To make things more interesting, we will add a twist: Only the elements that are not of type `ProtectedElement` will have their attributes converted into elements. Protected elements will be output as is. To this end, we assume that the type `ProtectedElements` is defined in a schema found in the document at the URL `http://types.r.us/auxiliary/types.xsd` with the target namespace `http://types.r.us/auxiliary`:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://types.r.us/auxiliary">
    <complexType name="ProtectedElement">
         ... ... ...
    </complexType>
</schema>
```

Figure 18.24 shows the XQuery expression we are looking for.

[25] Recall that `@*` is an XPath wildcard that returns all the attributes of an element. The XPath function `text()` returns all *t*-children of the current node.

[26] In this case, the element would have the form `<foo>some text</foo>`.

[27] For instance, `<foo>some text<bar>more text</bar>even more</foo>` has mixed content, which consists of two *t*-children and one *e*-child.

The first clause, SCHEMA, tells the XQuery processor where to find the schema used in the query (the URL following "at") and what is the namespace associated with that schema (the URL preceding "at"). In our case, this schema contains the definition of the union type `ProtectedElement`, which we use inside the function `convertNode()`. The DECLARE NAMESPACE clause introduces the namespace prefix `aux` for the namespace `http://types.r.us/auxiliary`, which is used in the query. This namespace identifies the schema where the type `ProtectedElement` is defined. Note that this namespace matches both the target namespace of the schema document that defines `ProtectedElement` and the namespace mentioned in the SCHEMA clause.

The function `convertNode()` takes an argument of type `node` and returns document nodes of type `node`. This is a built-in type of XQuery, which corresponds to any node in the document tree.

The only other new feature in the query is the predicate INSTANCEOF, which tests whether a given node (the value of `$n` in our case) conforms to a given type. We use this test in the function `convertNode()` to determine whether the argument to the function is an attribute node or an element of type `ProtectedElement`.

The function `convertNode()` works analogously to `convertElement()`, which we discussed earlier. It first checks the argument type. If it is an unprotected element, the appropriate element is created. Its contents is specified by two sequences of recursive calls to `convertNode()`. The first sequence converts the attributes of the current element into elements and the second convert each *et*-child of the current

Figure 18.24 XQuery transformation that does the same work as the stylesheet in Figure 18.18.

```
IMPORT SCHEMA "http://types.r.us/auxiliary
               at  http://types.r.us/auxiliary/types.xsd"
DECLARE NAMESPACE aux = "http://types.r.us/auxiliary"
DEFINE FUNCTION convertNode($n AS node) AS node {
     RETURN
          IF ($n NOT INSTANCEOF aux:ProtectedElement
               AND $n INSTANCEOF element) THEN {
               element {name($n)}
               {
                    {FOR $a IN $n/@* RETURN convertNode($a)},
                    {FOR $c IN $n/node() RETURN convertNode($c)}
               }
          } ELSE IF $n INSTANCEOF attribute THEN {
               element {name($n)} {data($n)}
          } ELSE $n
}

RETURN convertNode(document("....")/*)
```

node.[28] If the argument is an attribute, it is replaced with an element that has the same name as the attribute, and the attribute value becomes the contents of that element. If the argument is neither an attribute nor an element, it must be a text node, in which case the node is simply copied to the result document.

Grouping and aggregation. Unlike SQL, XQuery does not use a separate grouping operator; instead it relies on a more consistent mechanism that applies aggregate functions to explicitly constructed collections of document nodes. This is achieved with the help of the already familiar LET clause, which declares a variable and initializes it with a collection of nodes specified by an XQuery expression. Interestingly, the use of LET outside of a FOR clause is just syntactic sugar, as we saw in Figure 18.22, but its use in the scope of a FOR clause can be essential because grouping cannot be achieved in any other way.

To illustrate, the following query takes the Transcripts document in Figure 18.19 on page 664 and produces a document that lists students along with the number of courses each has taken so far.

```
FOR $t IN document("http://xyz.edu/transcripts.xml")//Transcript,
    $s IN $t/Student
LET $c := $t/CrsTaken
RETURN
    <StudentSummary StudId={$s/@StudId} Name={$s/@Name}
                    TotalCourses={count(distinct-values($c))}/>
ORDER BY StudentSummary/@TotalCourses
```

Recall that the FOR clause iterates by binding $t to every Transcript element of the document. For each such binding there is a unique Student element that gets bound to $s.[29] The trick here is that the variable $c is assigned a new list of CrsTaken elements each time $t is bound to a new element, because LET occurs in the scope of the FOR clause that binds $t. In other words, for any Transcript binding for $t, $c is bound to the list of classes mentioned in that transcript element and the subsequent call to count() simply counts the number of distinct elements on the list.

The similarity between the FOR and LET clauses might be deceiving. Both specify variable bindings as collections of document nodes, but FOR binds the variables to the *individual nodes* of a collection *in succession* while LET binds the variables to the *entire collection* of nodes *at once*.

The next query is slightly more complicated, as it involves a join of the Transcripts document in Figure 18.19 and the Classes document in Figure 18.21. It creates a list of classes along with the average grade in each. To obtain the numeric value for a grade, we use the function numericGrade(), which can be easily defined as an XQuery function (and is omitted). This example also illustrates another

[28] Recall that node() is an XPath function that returns all *et*-children of the current node.

[29] If a Transcript element could have several Student children, then $s would bind to each in succession and the FOR clause would function as a nested loop.

important feature: The binding for the variables in the LET clause (and, for that matter, in the FOR clause) does not need to be specified as an XPath expression, but can be any XQuery expression that returns a list of nodes. In particular, it can be a query as shown below:

```
FOR $c IN document("http://xyz.edu/classes.xml")//Class
-- $g gets the collection of all numeric grades in the class bound to $c
LET $g := {
            FOR $ct IN document("http://xyz.edu/transcripts.xml")
                                        //CrsTaken
            WHERE $ct/@CrsCode = $c/@CrsCode
                   AND $ct/@Semester = $c/@Semester
            RETURN numericGrade($ct/@Grade)
          }
RETURN
     <ClassSummary CrsCode = {$c/@CrsCode} Semester={$c/@Semester}
                   CsrName = {$c/CrsName} Instructor={$c/Instructor}
                   AvgGrade = {avg($g)}/>
ORDER BY ClassSummary/@CrsCode
```

This is essentially the same query as the one on page 666, which constructs class rosters by joining the documents `transcripts.xml` and `classes.xml`. However, instead of listing all students in the class in the result document, we compute the list of all grades in the class and assign it to a variable, `$g`, using the LET clause. These grades are then averaged, and the result is assigned as a value of the attribute `AvgGrade`.

Note that, when the LET clause occurs in the scope of a FOR clause, it introduces a new issue as far as the query semantics is concerned. This happens because now variables are bound both by the FOR and LET clauses. The LET clause is incorporated into the evaluation mechanism as follows. For each tuple of bindings for the variables in the FOR clause, the bindings for the LET variables are determined. Unlike the bindings for the FOR variables, the LET clause bindings are to lists of nodes, which are typically used as arguments to aggregate functions. Thus, the binding for the LET variables is completely determined by the binding for the FOR variables.

The rest of the query evaluation procedure remains unchanged: the WHERE clause filters out tuples of bindings produced by the FOR clause. The only difference is that now the condition in WHERE might also use the variables bound by LET. Finally, for each tuple of bindings for the FOR-variables, the RETURN clause generates a fragment for the result document.

Quantification. Suppose we want to extract the list of all students who have taken `MAT123`. In SQL, this requires the EXISTS operator (which is analogous to the XQuery function `empty()`). XQuery provides similar facilities through the quantifiers: SOME and EVERY, which precisely correspond to the existential quantifier $\exists$ and the universal quantifier $\forall$ in relational calculus (Chapter 6). As shown in

Chapter 6, explicit use of quantifiers greatly simplifies the formulation of certain queries as compared to their formulation in SQL.

The next query uses the SOME quantifier to return the list of all students who took MAT123.

```
FOR $t IN document("http://xyz.edu/transcripts.xml")//Transcript
WHERE SOME $ct IN $t/CrsTaken
          SATISFIES $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

In many cases, the SOME quantifier can be eliminated from the query. For instance assuming that a student cannot take the same course twice, the above query is equivalent to

```
FOR $t IN document("http://xyz.edu/transcripts.xml")//Transcript,
    $ct IN $t/CrsTaken
WHERE $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

The analogy between SQL and XQuery should not be taken too far, however. Consider the following queries, which return the course names for classes that have at least one enrolled student: The SQL query

```
SELECT   C.CrsName
FROM     CLASS C, TRANSCRIPT T
WHERE    C.CrsCode = T.CrsCode AND C.Semester = T.Semester
```

and the XQuery query

```
FOR      $c IN document("http://xyz.edu/classes.xml")//Class,
         $ct IN document("http://xyz.edu/transcripts.xml")//CrsTaken
WHERE    $c/@CrsCode = $ct/@CrsCode AND $c/@Semester = $ct/@Semester
RETURN   $c/CrsName
```

Since T does not occur in the SELECT clause and $t does not occur in the RETURN clause, the two are assumed to be existentially quantified. In both cases, a course name is output even if just one matching transcript record is found. If more than one is found, an SQL query processor might or might not output duplicate course names, depending on the inner workings of the query optimizer. (Since SQL is a relational query language, its semantics implies that only one name per course should be output. The possibility of duplicates is an implementation detail.) In contrast, the semantics of the FOR clause in XQuery implies that a course name will be output for *every* matching transcript record. This is because FOR specifies a loop in which the RETURN clause is executed for each binding of $c and $t, and every binding for $c is likely to have multiple matching bindings for $t.

Note that, while SELECT DISTINCT in SQL guarantees that there are no duplicates, the distinct-values() function in XQuery does not easily achieve the same goal

(see exercise 18.29). One way to eliminate duplicates in our example is to use SOME. Unlike the previous example of students who took MAT123, however, SOME is essential and cannot be eliminated simply by moving variables from the WHERE to the FOR clause.

```
FOR    $c IN document("http://xyz.edu/classes.xml")//Class
WHERE
  SOME $ct IN document("http://xyz.edu/transcripts.xml")//CrsTaken
  SATISFIES $c/@CrsCode = $ct/@CrsCode AND $c/@Semester = $ct/@Semester
RETURN $c/CrsName
```

In contrast to the existential quantifier, the universal quantifier, EVERY, cannot be easily eliminated from most queries. As discussed in Chapter 6, the universal quantifier provides a natural way of expressing queries that involve the division operator in relational algebra. Such queries look rather awkward in SQL, because SQL does not support universal quantification directly.[30] To illustrate, the following query retrieves all classes in which every enrolled student took MAT123:

```
FOR $c IN document("http://xyz.edu/classes.xml")//Class
-- $g gets bound to the set of all Transcript elements
-- corresponding to the particular class that binds $c
LET $g := {
     FOR $t IN document("http://xyz.edu/transcripts.xml")
                                //Transcript
     WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and
                       @Semester = $c/@Semester]
     RETURN $t
           }
-- Take only those $g in which every transcript
-- has a CrsTaken element for MAT123
WHERE EVERY $tr IN $g
     SATISFIES NOT empty($tr[CrsTaken/@CrsCode = "MAT123"])
RETURN $c ORDER BY $c/@CrsCode
```

Here, for every binding of $c to a Class element, $g is bound to the list of transcripts of students who took this class. The outer WHERE clause checks that every student transcript in $g indicates that the student has taken MAT123. (Recall that the XPath expression $tr[CrsTaken/@CrsCode = "MAT123"] returns a nonempty set of nodes if and only if $tr is bound to a transcript element that includes a CrsTaken element for the course MAT123.)

[30] We saw in Section 5.2.3 that expressing universal quantification in SQL requires EXISTS, nested subqueries, and double negation. However, SQL:1999 introduces a limited form of universal quantification, the FOR ALL operator, which provides relief for writing many queries that require the division operator of relational algebra.