CHAPTER 30

XML

The **Extensible Markup Language** (**XML**) was not designed for database applications. In fact, like the **Hyper-Text Markup Language** (**HTML**) on which the World Wide Web is based, XML has its roots in document management and is derived from a language for structuring large documents known as the **Standard Generalized Markup Language** (**SGML**). However, unlike SGML and HTML, XML is designed to represent data. It is particularly useful as a data format when an application must communicate with another application or integrate information from several other applications. When XML is used in these contexts, many database issues arise, including how to organize, manipulate, and query the XML data. In this chapter, we introduce XML and discuss both the management of XML data with database techniques and the exchange of data formatted as XML documents.

30.1 Motivation

To understand XML, it is important to understand its roots as a document markup language. The term **markup** refers to anything in a document that is not intended to be part of the printed output. For example, a writer creating text that will eventually be typeset in a magazine may want to make notes about how the typesetting should be done. It would be important to type these notes in a way that they could be distinguished from the actual content, so that a note like "set this word in large size, bold font" or "insert a line break here" does not end up printed in the magazine. Such notes convey extra information about the text. In electronic document processing, a **markup language** is a formal description of what part of the document is content, what part is markup, and what the markup means.

Just as database systems evolved from physical file processing to provide a separate logical view, markup languages evolved from specifying instructions for how to print parts of the document to specifying the *function* of the content. For instance, with functional markup, text representing section headings (for this section, the word *Motivation* would be marked up as being a section heading, instead of being marked up as text

to be printed in large size, bold font). From the viewpoint of typesetting, such functional markup allows the document to be formatted differently in different situations. It also helps different parts of a large document, or different pages in a large web site, to be formatted in a uniform manner. More importantly, functional markup also helps record what each part of the text represents semantically, and correspondingly helps automate extraction of key parts of documents.

For the family of markup languages that includes HTML, SGML, and XML, the markup takes the form of **tags** enclosed in angle brackets, <>. Tags are used in pairs, with <tag> and </tag> delimiting the beginning and the end of the portion of the document to which the tag refers. For example, the title of a document might be marked up as follows:

```
<title>Database System Concepts</title>
```

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course>
        <course_id> CS-101 </course_id>
        <title> Intro. to Computer Science </title>
        <dept_name> Comp. Sci </dept_name>
        <credits> 4 </credits>
    </course>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
```

continued in Figure Figure 30.2

Figure 30.1 XML representation of (part of) university information.

Unlike HTML, XML does not prescribe the set of tags allowed, and the set may be chosen as needed by each application. This feature is the key to XML's major role in data representation and exchange, whereas HTML is used primarily for document formatting.

For example, in our running university application, department, course and instructor information can be represented as part of an XML document as in Figure 30.1 and Figure 30.2. Observe the use of tags such as department, course, instructor, and teaches. To keep the example short, we use a simplified version of the university

```
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000 </salary>
    </instructor>
    <instructor>
        <IID> 83821 </IID>
        <name> Brandt </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 92000 </salary>
    </instructor>
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 83821 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 76766 </IID>
        <course_id> BIO-301 </course_id>
    </teaches>
</university>
```

Figure 30.2 Continuation of Figure 30.1.

schema that ignores section information for courses. We have also used the tag IID to denote the identifier of the instructor, for reasons we shall see later.

These tags provide context for each value and allow the semantics of the value to be identified. For this example, the XML data representation does not provide any significant benefit over the traditional relational data representation; however, we use this example as our running example because of its simplicity.

Figure 30.3, which shows how information about a purchase order can be represented in XML, illustrates a more realistic use of XML. Purchase orders are typically generated by one organization and sent to another. Traditionally they were printed on paper by the purchaser and sent to the supplier; the data would be manually re-entered

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA  </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies  </name>
        <address> 1 Broadway, New York, NY, USA  </address>
    </supplier>
    <itemlist>
        <item>
            <identifier>  RS1  </identifier>
            <description> Atom powered rocket sled  </description>
            <quantity> 2 </quantity>
            <price> 199.95 </price>
        </item>
        <item>
            <identifier>  SG2  </identifier>
            <description> Superb glue  </description>
            <quantity> 1 </quantity>
            <unit-of-measure> liter </unit-of-measure>
            <price> 29.95 </price>
        </item>
    </itemlist>
    <total_cost> 429.85 </total_cost>
    <payment_terms> Cash-on-delivery </payment_terms>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchaseorder>
```

Figure 30.3 XML representation of a purchase order.

into a computer system by the supplier. This slow process can be greatly sped up by sending the information electronically between the purchaser and supplier. The nested representation allows all information in a purchase order to be represented naturally in a single document. (Real purchase orders have considerably more information than that depicted in this simplified example.) XML provides a standard way of tagging the data; the two organizations must agree on what tags appear in the purchase order, and what they mean.

Compared to storage of data in a relational database, the XML representation may be inefficient, since tag names are repeated throughout the document. However, in spite of this disadvantage, an XML representation has significant advantages when it is used to exchange data between organizations, and for storing complex structured information in files:

- First, the presence of the tags makes the message **self-documenting**; that is, a schema need not be consulted to understand the meaning of the text. We can readily read the fragment in Figure 30.1 and Figure 30.2, for example.
- Second, the format of the document is not rigid. For example, if some sender adds additional information, such as a tag last_accessed noting the last date on which an account was accessed, the recipient of the XML data may simply ignore the tag. As another example, in Figure 30.3, the item with identifier SG2 has a tag called unit-of-measure specified, which the first item does not. The tag is required for items that are ordered by weight or volume and may be omitted for items that are simply ordered by number.

  The ability to recognize and ignore unexpected tags allows the format of the data to evolve over time, without invalidating existing applications. Similarly, the ability to have multiple occurrences of the same tag makes it easy to represent multivalued attributes.
- Third, XML allows nested structures. The purchase order shown in Figure 30.3 illustrates the benefits of having a nested structure. Each purchase order has a purchaser and a list of items as two of its nested structures. Each item in turn has an item identifier, description, and a price nested within it, while the purchaser has a name and address nested within it.

  Such information would have been split into multiple relations in a relational schema. Item information would have been stored in one relation, purchaser information in a second relation, purchase orders in a third, and the relationship between purchase orders, purchasers, and items would have been stored in a fourth relation.

  The relational representation helps to avoid redundancy; for example, item descriptions would be stored only once for each item identifier in a normalized relational schema. In the XML purchase order, however, the descriptions may be repeated in multiple purchase orders that order the same item. However, gathering all information related to a purchase order into a single nested structure, even at

the cost of redundancy, is attractive when information has to be exchanged with external parties.

- Finally, since the XML format is widely accepted, a wide variety of tools are available to assist in its processing, including programming language APIs to create and to read XML data, browser software, and database tools.

We describe several applications for XML data in Section 30.7. Just as SQL is the dominant *language* for querying relational data, XML has become the dominant *format* for data exchange.

30.2 Structure of XML Data

The fundamental construct in an XML document is the **element**. An element is simply a pair of matching start- and end-tags and all the text that appears between them.

XML documents must have a single **root element** that encompasses all other elements in the document. In the example in Figure 30.1, the <university> element forms the root element. Further, elements in an XML document must **nest** properly. For instance,

<course> … <title> … </title> … </course>

is properly nested, whereas

<course> … <title> … </course> … </title>

is not properly nested.

While proper nesting is an intuitive property, we may define it more formally. Text is said to appear **in the context of** an element if it appears between the start-tag and end-

```
…
    <course>
        This course is being offered for the first time in 2009.
        <course_id> BIO-399 </course_id>
        <title> Computational Biology </title>
        <dept_name> Biology </dept_name>
        <credits> 3 </credits>
    </course>
…
```

Figure 30.4 Mixture of text with subelements.

tag of that element. Tags are properly nested if every start-tag has a unique matching end-tag that is in the context of the same parent element.

Note that text may be mixed with the subelements of an element, as in Figure 30.4. As with several other features of XML, this freedom makes more sense in a document-processing context than in a data-processing context, and it is not particularly useful for representing more-structured data such as database content in XML.

```
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000. </salary>
        <course_id> CS-101 </course_id>
    </instructor>
</university-1>
```

Figure 30.5 Nested XML representation of university information.

```
<university-2>
    <instructor>
        <ID> 10101 </ID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 65000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>

    <instructor>
        <ID> 83821 </ID>
        <name> Brandt </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 92000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>
</university-2>
```

Figure 30.6 Redundancy in nested XML representation.

The ability to nest elements within other elements provides an alternative way to represent information. Figure 30.5 shows a representation of part of the university information from Figure 30.1, but with course elements nested within department elements. The nested representation makes it easy to find all courses offered by a department. Similarly, identifiers of courses taught by an instructor are nested within the instructor elements. If an instructor teaches more than one course, there would be multiple course_id elements within the corresponding instructor element. Details of

instructors Brandt and Crick are omitted from Figure 30.5 for lack of space but are similar in structure to that for Srinivasan.

Although nested representations are natural in XML, they may lead to redundant storage of data. For example, suppose details of courses taught by an instructor are stored nested within the instructor element as shown in Figure 30.6. If a course is taught by more than one instructor, course information such as title, department, and credits would be stored redundantly with every instructor associated with the course.

Nested representations are widely used in XML data interchange applications to avoid joins. For instance, a purchase order would store the full address of sender and receiver redundantly on multiple purchase orders, whereas a normalized representation may require a join of purchase order records with a *company_address* relation to get address information.

In addition to elements, XML specifies the notion of an **attribute**. For instance, the course identifier of a course can be represented as an attribute, as shown in Figure 30.7. The attributes of an element appear as *name=value* pairs before the closing ">" of a tag. Attributes are strings and do not contain markup. Furthermore, attributes can appear only once in a given tag, unlike subelements, which may be repeated.

Note that in a document construction context, the distinction between subelement and attribute is important—an attribute is implicitly text that does not appear in the printed or displayed document. However, in database and data exchange applications of XML, this distinction is less relevant, and the choice of representing data as an attribute or a subelement is frequently arbitrary. In general, it is advisable to use attributes only to represent identifiers, and to store all other data as subelements.

One final syntactic note is that an element of the form <element></element> that contains no subelements or text can be abbreviated as <element/>; abbreviated elements may, however, contain attributes.

Since XML documents are designed to be exchanged between applications, a **namespace** mechanism has been introduced to allow organizations to specify globally unique names to be used as element tags in documents. The idea of a namespace is to prepend each tag or attribute with a universal resource identifier (e.g., a web address). Thus, for example, if Yale University wanted to ensure that XML documents it created would

```
...
    <course course_id= "CS-101">
        <title> Intro. to Computer Science</title>
        <dept_name> Comp. Sci. </dept_name>
        <credits> 4 </credits>
    </course>
...
```

Figure 30.7 Use of attributes.

```
<university xmlns:yale="http://www.yale.edu">
    ...
    <yale:course>
        <yale:course_id> CS-101 </yale:course_id>
        <yale:title> Intro. to Computer Science</yale:title>
        <yale:dept_name> Comp. Sci. </yale:dept_name>
        <yale:credits> 4 </yale:credits>
    </yale:course>
    ...
</university>
```

Figure 30.8 Unique tag names can be assigned by using namespaces.

not duplicate tags used by any business partner's XML documents, it could prepend a unique identifier with a colon to each tag name. The university may use a web URL such as

http://www.yale.edu

as a unique identifier. Using long unique identifiers in every tag would be rather inconvenient, so the namespace standard provides a way to define an abbreviation for identifiers.

In Figure 30.8, the root element (university) has an attribute xmlns:yale, which declares that yale is defined as an abbreviation for the URL given above. The abbreviation can then be used in various element tags, as illustrated in the figure.

A document can have more than one namespace, declared as part of the root element. Different elements can then be associated with different namespaces. A **default namespace** can be defined by using the attribute xmlns instead of xmlns:yale in the root element. Elements without an explicit namespace prefix would then belong to the default namespace.

Sometimes we need to store values containing tags without having the tags interpreted as XML tags. So that we can do so, XML allows this construct:

<![CDATA[<course> ···</course>]]>

Because it is enclosed within CDATA, the text <course> is treated as normal text data, not as a tag. The term *CDATA* stands for character data.

30.3 XML Document Schema

Databases have schemas, which are used to constrain what information can be stored in the database and to constrain the data types of the stored information. In contrast,

```
<!DOCTYPE university [
    <!ELEMENT university ( (department|course|instructor|teaches)+)>
    <!ELEMENT department ( dept_name, building, budget)>
    <!ELEMENT course ( course_id, title, dept_name, credits)>
    <!ELEMENT instructor (IID, name, dept_name, salary)>
    <!ELEMENT teaches (IID, course_id)>
    <!ELEMENT dept_name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course_id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
    <!ELEMENT IID( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT salary( #PCDATA )>
] >
```

Figure 30.9 Example of a DTD.

by default, XML documents can be created without any associated schema: an element may then have any subelement or attribute. While such freedom may occasionally be acceptable given the self-describing nature of the data format, it is not generally useful when XML documents must be processed automatically as part of an application, or even when large amounts of related data are to be formatted in XML.

Here, we describe the first **schema-definition** language included as part of the XML standard, the *document type definition*, as well as its more recently defined replacement, *XML Schema*. Another XML schema-definition language called Relax NG is also in use, but we do not cover it here; for more information on Relax NG, see the references in the bibliographical notes section.

30.3.1 Document Type Definition

The **document type definition** (**DTD**) is an optional part of an XML document. The main purpose of a DTD is much like that of a schema: to constrain and type the information present in the document. However, the DTD does not in fact constrain types in the sense of basic types like integer or string. Instead, it constrains only the appearance of subelements and attributes within an element. The DTD is primarily a list of rules for what pattern of subelements may appear within an element. Figure 30.9 shows a part of an example DTD for a university information document; the XML document in Figure 30.1 conforms to this DTD.

Each declaration is in the form of a regular expression for the subelements of an element. Thus, in the DTD in Figure 30.9, a university element consists of one or more course, department, or instructor elements; the | operator specifies "or" while the +

operator specifies "one or more." Although not shown here, the ∗ operator is used to specify "zero or more," while the ? operator is used to specify an optional element (i.e., "zero or one").

The course element contains subelements course_id, title, dept_name, and credits (in that order). Similarly, department and instructor have the attributes of their relational schema defined as subelements in the DTD.

Finally, the elements course_id, title, dept_name, credits, building, budget, IID, name, and salary are all declared to be of type #PCDATA. The keyword #PCDATA indicates text data; it derives its name, historically, from "parsed character data." Two other special type declarations are empty, which says that the element has no contents, and any, which says that there is no constraint on the subelements of the element; that is, any elements, even those not mentioned in the DTD, can occur as subelements of the element. The absence of a declaration for an element is equivalent to explicitly declaring the type as any.

The allowable attributes for each element are also declared in the DTD. Unlike subelements, no order is imposed on attributes. Attributes may be specified to be of type CDATA, ID, IDREF, or IDREFS; the type CDATA simply says that the attribute contains character data, while the other three are not so simple; they are explained in more detail shortly. For instance, the following line from a DTD specifies that element course has an attribute of type course_id, and a value must be present for this attribute:

```
<!ATTLIST course course_id CDATA #REQUIRED>
```

Attributes must have a type declaration and a default declaration. The default declaration can consist of a default value for the attribute or #REQUIRED, meaning that a value must be specified for the attribute in each element, or #IMPLIED, meaning that no default value has been provided, and the document may omit this attribute. If an attribute has a default value, for every element that does not specify a value for the attribute, the default value is filled in automatically when the XML document is read.

An attribute of type ID provides a unique identifier for the element; a value that occurs in an ID attribute of an element must not occur in any other element in the same document. At most one attribute of an element is permitted to be of type ID. (We renamed the attribute *ID* of the *instructor* relation to IID in the XML representation, in order to avoid confusion with the type ID.)

An attribute of type IDREF is a reference to an element; the attribute must contain a value that appears in the ID attribute of some element in the document. The type IDREFS allows a list of references, separated by spaces.

Figure 30.10 shows an example DTD in which identifiers of course, department, and instructor are represented by ID attributes, and relationships between them are represented by IDREF and IDREFS attributes. The course elements use course_id as their identifier attribute; to do so, course_id has been made an attribute of course instead of a subelement. Additionally, each course element also contains an IDREF of the department corresponding to the course and an IDREFS attribute instructors identifying

```
<!DOCTYPE university-3 [
    <!ELEMENT university ( (department|course|instructor)+)>
    <!ELEMENT department ( building, budget )>
    <!ATTLIST department
        dept_name ID #REQUIRED >
    <!ELEMENT course (title, credits )>
    <!ATTLIST course
        course_id ID #REQUIRED
        dept_name IDREF #REQUIRED
        instructors IDREFS #IMPLIED >
    <!ELEMENT instructor ( name, salary )>
    <!ATTLIST instructor
        IID ID #REQUIRED
        dept_name IDREF #REQUIRED >
    ··· declarations for title, credits, building,
        budget, name and salary ···
] >
```

Figure 30.10 DTD with ID and IDREFS attribute types.

the instructors who teach the course. The department elements have an identifier attribute called dept_name. The instructor elements have an identifier attribute called IID and an IDREF attribute dept_name identifying the department to which the instructor belongs.

Figure 30.11 shows an example XML document based on the DTD in Figure 30.10.

The ID and IDREF attributes serve the same role as reference mechanisms in object-oriented and object-relational databases, permitting the construction of complex data relationships.

Document type definitions are strongly connected to the document formatting heritage of XML. Because of this, they are unsuitable in many ways for serving as the type structure of XML for data-processing applications. Nevertheless, a number of data exchange formats have been defined in terms of DTDs, since they were part of the original standard. Here are some of the limitations of DTDs as a schema mechanism:

- Individual text elements and attributes cannot be typed further. For instance, the element balance cannot be constrained to be a positive number. The lack of such constraints is problematic for data processing and exchange applications, which must then contain code to verify the types of elements and attributes.
- It is difficult to use the DTD mechanism to specify unordered sets of subelements. Order is seldom important for data exchange (unlike document layout, where it is crucial). While the combination of alternation (the | operation) and the $*$ or the

```
<university-3>
    <department dept_name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept_name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course_id="CS-101" dept_name="Comp. Sci"
                                instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    <course course_id="BIO-301" dept_name="Biology"
                                    instructors="76766">
        <title> Genetics </title>
        <credits> 4 </credits>
    </course>
    <instructor IID="10101" dept_name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    <instructor IID="83821" dept_name="Comp. Sci.">
        <name> Brandt </name>
        <salary> 72000 </salary>
    </instructor>
    <instructor IID="76766" dept_name="Biology">
        <name> Crick </name>
        <salary> 72000 </salary>
    </instructor>
</university-3>
```

Figure 30.11 XML data with ID and IDREF attributes.

+ operation as in Figure 30.9 permits the specification of unordered collections of tags, it is much more difficult to specify that each tag may only appear once.

- There is a lack of typing in IDs and IDREFSs. Thus, there is no way to specify the type of element to which an IDREF or IDREFS attribute should refer. As a result, the DTD in Figure 30.10 does not prevent the "dept_name" attribute of a course element from referring to other courses, even though this makes no sense.

30.3.2 XML Schema

An effort to redress the deficiencies of the DTD mechanism resulted in the development of a more sophisticated schema language, **XML Schema**. We provide a brief overview of XML Schema, and then we list some areas in which it improves DTDs.

XML Schema defines a number of built-in types such as string, integer, decimal date, and boolean. In addition, it allows user-defined types; these may be simple types

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="course">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="course_id" type="xs:string"/>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="credits" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

continued in Figure 30.13.

Figure 30.12 XML Schema version of DTD from Figure 30.9.

```
<xs:element name="teaches">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="course_id" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="UniversityType">
    <xs:sequence>
        <xs:element ref="department" minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element ref="course" minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element ref="instructor" minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element ref="teaches" minOccurs="0"
                    maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Figure 30.13 Continuation of Figure 30.12.

with added restrictions, or complex types constructed using constructors such as complexType and sequence.

Figure 30.12 and Figure 30.13 show how the DTD in Figure 30.9 can be represented by XML Schema; we describe next XML Schema features illustrated by the figures.

The first thing to note is that schema definitions in XML Schema are themselves specified in XML syntax, using a variety of tags defined by XML Schema. To avoid conflicts with user-defined tags, we prefix the XML Schema tag with the namespace prefix "xs:"; this prefix is associated with the XML Schema namespace by the xmlns:xs specification in the root element:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Note that any namespace prefix could be used in place of xs; thus, we could replace all occurrences of "xs:" in the schema definition with "xsd:" without changing the meaning of the schema definition. All types defined by XML Schema must be prefixed by this namespace prefix.

The first element is the root element university, whose type is specified to be UniversityType, which is declared later. The example then defines the types of elements department, course, instructor, and teaches.

Note that each of these is specified by an element with tag xs:element, whose body contains the type definition.

The type of department is defined to be a complex type, which is further specified to consist of a sequence of elements dept_name, building, and budget. Any type that has either attributes or nested subelements must be specified to be a complex type.

Alternatively, the type of an element can be specified to be a predefined type by the attribute type; observe how the XML Schema types xs:string and xs:decimal are used to constrain the types of data elements such as dept_name and credits.

Finally the example defines the type UniversityType as containing zero or more occurrences of each of department, course, instructor, and teaches. Note the use of ref to specify the occurrence of an element defined earlier. XML Schema can define the minimum and maximum number of occurrences of subelements by using minOccurs and maxOccurs. The default for both minimum and maximum occurrences is 1, so these have to be specified explicitly to allow zero or more department, course, instructor, and teaches elements.

Attributes are specified using the xs:attribute tag. For example, we could have defined dept_name as an attribute by adding

```
<xs:attribute name = "dept_name"/>
```

within the declaration of the department element. Adding the attribute use = "required" to the above attribute specification declares that the attribute must be specified, whereas the default value of use is optional. Attribute specifications would appear directly under the enclosing complexType specification, even if elements are nested within a sequence specification.

We can use the xs:complexType element to create named complex types; the syntax is the same as that used for the xs:complexType element in Figure 30.12, except that we add an attribute name = typeName to the xs:complexType element, where typeName is the name we wish to give to the type. We can then use the named type to specify the type of an element using the type attribute, just as we used xs:decimal and xs:string in our example.

In addition to defining types, a relational schema also allows the specification of constraints. XML Schema allows the specification of keys and key references, corresponding to the primary-key and foreign-key definition in SQL. In SQL, a primary-key constraint or unique constraint ensures that the attribute values do not recur within the relation. In the context of XML, we need to specify a scope within which values are unique and form a key. The selector is a path expression that defines the scope for the constraint, and field declarations specify the elements or attributes that form the key.[1] To specify that dept_name forms a key for department elements under the root university element, we add the following constraint specification to the schema definition:

[1] We use simple path expressions here that are in a familiar syntax. XML has a rich syntax for path expressions, called XPath, which we explore in Section 30.4.2.

```
<xs:key name = "deptKey">
    <xs:selector xpath = "/university/department"/>
    <xs:field xpath = "dept_name"/>
</xs:key>
```

Correspondingly a foreign-key constraint from course to department may be defined as follows:

```
<xs: name = "courseDeptFKey" refer="deptKey">
    <xs:selector xpath = "/university/course"/>
    <xs:field xpath = "dept_name"/>
</xs:keyref>
```

Note that the refer attribute specifies the name of the key declaration that is being referenced, while the field specification identifies the referring attributes.

XML Schema offers several benefits over DTDs and is widely used today. Among the benefits that we have seen in the preceding examples are these:

- It allows the text that appears in elements to be constrained to specific types, such as numeric types in specific formats or complex types such as sequences of elements of other types.
- It allows user-defined types to be created.
- It allows uniqueness and foreign-key constraints.
- It is integrated with namespaces to allow different parts of a document to conform to different schemas.

In addition to the features we have seen, XML Schema supports several other features that DTDs do not, such as these:

- It allows types to be restricted to create specialized types, for instance by specifying minimum and maximum values.
- It allows complex types to be extended by using a form of inheritance.

Our description of XML Schema is just an overview; to learn more about XML Schema, see the references in the bibliographical notes.

30.4 Querying and Transformation

Given the increasing number of applications that use XML to exchange, mediate, and store data, tools for effective management of XML data are becoming increasingly important. In particular, tools for querying and transformation of XML data are essential

to extract information from large bodies of XML data and to convert data between different representations (schemas) in XML. Just as the output of a relational query is a relation, the output of an XML query can be an XML document. As a result, querying and transformation can be combined into a single tool.

In this section, we describe the XPath and XQuery languages:

- **XPath** is a language for path expressions and is actually a building block for XQuery.
- **XQuery** is the standard language for querying XML data. It is modeled after SQL but is significantly different, since it has to deal with nested XML data. XQuery also incorporates XPath expressions.

The XSLT language is another language designed for transforming XML. However, it is used primarily in document-formatting applications, rather in data-management applications, so we do not discuss it in this book.

The tools section at the end of this chapter provides references to software that can be used to execute queries written in XPath and XQuery.

30.4.1 Tree Model of XML

A **tree model** of XML data are used in all these languages. An XML document is modeled as a **tree**, with **nodes** corresponding to elements and attributes. Element nodes can have child nodes, which can be subelements or attributes of the element. Correspondingly, each node (whether attribute or element), other than the root element, has a parent node, which is an element. The order of elements and attributes in the XML document is modeled by the ordering of children of nodes of the tree. The terms *parent*, *child*, *ancestor*, *descendant*, and *siblings* are used in the tree model of XML data.

The text content of an element can be modeled as a text-node child of the element. Elements containing text broken up by intervening subelements can have multiple text-node children. For instance, an element containing "this is a <bold> wonderful </bold> book" would have a subelement child corresponding to the element bold and two text node children corresponding to "this is a" and "book." Since such structures are not commonly used in data representation, we shall assume that elements do not contain both text and subelements.

30.4.2 XPath

XPath addresses parts of an XML document by means of path expressions. The language can be viewed as an extension of the simple path expressions in object-oriented and object-relational databases. The current version of the XPath standard is XPath 2.0, and our description is based on this version.

A **path expression** in XPath is a sequence of location steps separated by "/" (instead of the "." operator that separates location steps in SQL). The result of a path expression is a set of nodes. For instance, on the document in Figure 30.11, the XPath expression

```
/university-3/instructor/name
```

returns these elements:

```
<name>Srinivasan</name>
<name>Brandt</name>
```

The expression

```
/university-3/instructor/name/text()
```

returns the same names, but without the enclosing tags.

Path expressions are evaluated from left to right. Like a directory hierarchy, the initial '/' indicates the root of the document. Note that this is an abstract root "above" <university-3> that is the document tag.

As a path expression is evaluated, the result of the path at any point consists of an ordered set of nodes from the document. Initially, the "current" set of elements contains only one node, the abstract root. When the next step in a path expression is an element name, such as instructor, the result of the step consists of the nodes corresponding to elements of the specified name that are children of elements in the current element set. These nodes then become the current element set for the next step of the path expression evaluation. Thus, the expression

```
/university-3
```

returns a single node corresponding to the

```
<university-3>
```

tag, while

```
/university-3/instructor
```

returns the two nodes corresponding to the

```
instructor
```

elements that are children of the

```
university-3
```

node.

The result of a path expression is then the set of nodes after the last step of path expression evaluation. The nodes returned by each step appear in the same order as their appearance in the document.

Since multiple children can have the same name, the number of nodes in the node set can increase or decrease with each step. Attribute values may also be accessed, using the "@" symbol. For instance, /university-3/course/@course_id returns a set of all values of course_id attributes of course elements. By default, IDREF links are not followed; we shall see how to deal with IDREFs later.

XPath supports a number of other features:

- Selection predicates may follow any step in a path and are contained in square brackets. For example,

  /university-3/course[credits >= 4]

  returns course elements with a credits value greater than or equal to 4, while

  /university-3/course[credits >= 4]/@course_id

  returns the course identifiers of those courses.

  We can test the existence of a subelement by listing it without any comparison operation; for instance, if we removed just ">= 4" from the above, the expression would return course identifiers of all courses that have a credits subelement, regardless of its value.

- XPath provides several functions that can be used as part of predicates, including testing the position of the current node in the sibling order and the aggregate function count(), which counts the number of nodes matched by the expression to which it is applied. For example, on the XML representation in Figure 30.6, the path expression

  /university-2/instructor[count(./teaches/course)> 2]

  returns instructors who teach more than two courses. Boolean connectives and and or can be used in predicates, while the function not(…) can be used for negation.

- The function id("foo") returns the node (if any) with an attribute of type ID and value "foo". The function id can even be applied on sets of references, or even strings containing multiple references separated by blanks, such as IDREFS. For instance, the path

  /university-3/course/id(@dept_name)

returns all department elements referred to from the dept_name attribute of course elements, while

```
/university-3/course/id(@instructors)
```

returns the instructor elements referred to in the instructors attribute of course elements.

- The | operator allows expression results to be unioned. For example, given data using the schema from Figure 30.11, we could find the union of Computer Science and Biology courses using the expression:

```
/university-3/course[@dept_name="Comp. Sci"]  |
    /university-3/course[@dept_name="Biology"]
```

However, the | operator cannot be nested inside other operators. It is also worth noting that the nodes in the union are returned in the order in which they appear in the document.

- An XPath expression can skip multiple levels of nodes by using "//". For instance, the expression /university-3//name finds all name elements *anywhere* under the /university-3 element, regardless of the elements in which they are contained, and regardless of how many levels of enclosing elements are present between the university-3 and name elements. This example illustrates the ability to find required data without full knowledge of the schema.

- A step in the path need not just select from the children of the nodes in the current node set. In fact, this is just one of several directions along which a step in the path may proceed, such as parents, siblings, ancestors, and descendants. We omit details, but note that "//", described above, is a short form for specifying "all descendants," while ".." specifies the parent.

- The built-in function doc(name) returns the root of a named document; the name could be a file name or a URL. The root returned by the function can then be used in a path expression to access the contents of the document. Thus, a path expression can be applied on a specified document, instead of being applied on the current default document.

  For example, if the university data in our university example are contained in a file "university.xml", the following path expression would return all departments at the university:

```
doc("university.xml")/university/department
```

The function collection(name) is similar to doc but returns a collection of documents identified by name. The function collection can be used, for example, to

open an XML database, which can be viewed as a collection of documents; the following element in the XPath expression would select the appropriate document(s) from the collection.

In most of our examples, we assume that the expressions are evaluated in the context of a database, which implicitly provides a collection of "documents" on which XPath expressions are evaluated. In such cases, we do not need to use the functions doc and collection.

30.4.3 XQuery

The World Wide Web Consortium (W3C) has developed XQuery as the standard query language for XML. Our discussion is based on XQuery 1.0, which was released as a W3C recommendation on 23 January 2007.

30.4.3.1 FLWOR Expressions

XQuery queries are modeled after SQL queries but differ significantly from SQL. They are organized into five sections: **for**, **let**, **where**, **order by**, and **return**. They are referred to as "FLWOR" (pronounced "flower") expressions, with the letters in FLWOR denoting the five sections.

A simple FLWOR expression that returns course identifiers of courses with greater than three credits, shown below, is based on the XML document of Figure 30.11, which uses ID and IDREFS:

```
for $x in /university-3/course
let $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course_id>
```

The **for** clause is like the **from** clause of SQL and specifies variables that range over the results of XPath expressions. When more than one variable is specified, the results include the Cartesian product of the possible values the variables can take, just as the SQL **from** clause does.

The **let** clause simply allows the results of XPath expressions to be assigned to variable names for simplicity of representation. The **where** clause, like the SQL **where** clause, performs additional tests on the joined tuples from the **for** clause. The **order by** clause, like the SQL **order by** clause, allows sorting of the output. Finally, the **return** clause allows the construction of results in XML.

A FLWOR query need not contain all the clauses; for example a query may contain just the **for** and **return** clauses, and omit the **let**, **where**, and **order by** clauses. The preceding XQuery query did not contain an **order by** clause. In fact, since this query is simple, we can easily do away with the **let** clause, and the variable $courseId in the **return** clause could be replaced with $x/@course_id. Note further that, since the **for** clause uses XPath expressions, selections may occur within the XPath expression. Thus, an equivalent query may have only **for** and **return** clauses:

```
for $x in /university-3/course[credits > 3]
return <course_id> { $x/@course_id } </course_id>
```

However, the **let** clause helps simplify complex queries. Note also that variables assigned by **let** clauses may contain sequences with multiple elements or values, if the path expression on the right-hand side returns a sequence of multiple elements or values.

Observe the use of curly brackets ("{}") in the **return** clause. When XQuery finds an element such as <course_id> starting an expression, it treats its contents as regular XML text, except for portions enclosed within curly brackets, which are evaluated as expressions. Thus, if we omitted the curly brackets in the above **return** clause, the result would contain several copies of the string "$x/@course_id" each enclosed in a course_id tag. The contents within the curly brackets are, however, treated as expressions to be evaluated. Note that this convention applies even if the curly brackets appear within quotes. Thus, we could modify the preceding query to return an element with tag course, with the course identifier as an attribute, by replacing the **return** clause with the following:

```
return <course course_id="{$x/@course_id}" />
```

XQuery provides another way of constructing elements using the **element** and **attribute** constructors. For example, if the **return** clause in the previous query is replaced by the following **return** clause, the query would return course elements with course_id and dept_name as attributes and title and credits as subelements.

```
return element course {
        attribute course_id {$x/@course_id},
        attribute dept_name {$x/dept_name},
        element title {$x/title},
        element credits {$x/credits}
}
```

Note that, as before, the curly brackets are required to treat a string as an expression to be evaluated.

30.4.3.2 Joins

Joins are specified in XQuery much as they are in SQL. The join of course, instructor, and teaches elements in Figure 30.1 can be written in XQuery this way:

```
for $c in /university/course,
      $i in /university/instructor,
      $t in /university/teaches
where $c/course_id= $t/course_id
      and $t/IID = $i/IID
return <course_instructor> { $c $i } </course_instructor>
```

The same query can be expressed with the selections specified as XPath selections:

```
for $c in /university/course,
      $i in /university/instructor,
      $t in /university/teaches[ $c/course_id= $t/course_id
           and $t/IID = $i/IID]
return <course_instructor> { $c $i } </course_instructor>
```

Path expressions in XQuery are the same as path expressions in XPath 2.0. Path expressions may return a single value or element, or a sequence of values or elements. In the absence of schema information, it may not be possible to infer whether a path expression returns a single value or a sequence of values. Such path expressions may participate in comparison operations such as =, <, and >=.

XQuery has an interesting definition of comparison operations on sequences. For example, the expression `$x/credits > 3` would have the usual interpretation if the result of `$x/credits` is a single value, but if the result is a sequence containing multiple values, the expression evaluates to true if at least one of the values is greater than 3. Similarly, the expression `$x/credits = $y/credits` evaluates to true if any one of the values returned by the first expression is equal to any one of the values returned by the second expression. If this behavior is not appropriate, the operators `eq, ne, lt, gt, le, ge` can be used instead. These raise an error if either of their inputs is a sequence with multiple values.

30.4.3.3 Nested Queries

XQuery FLWOR expressions can be nested in the **return** clause in order to generate element nestings that do not appear in the source document. For instance, the XML structure shown in Figure 30.5, with course elements nested within department elements, can be generated from the structure in Figure 30.1 by the query shown in Figure 30.14.

The query also introduces the syntax `$d/*`, which refers to all the children of the node (or sequence of nodes) bound to the variable $d. Similarly, `$d/text()` gives the text content of an element, without the tags.

XQuery provides a variety of aggregate functions such as `sum()` and `count()` that can be applied on sequences of elements or values. The function `distinct-values()` applied on a sequence returns a sequence without duplication. The sequence (collection)

```
<university-1>
{
    for $d in /university/department
    return
        <department>
            { $d/* }
            { for $c in /university/course[dept_name = $d/dept_name]
              return $c }
        </department>
}
{
    for $i in /university/instructor
    return
        <instructor>
            { $i/* }
            { for $c in /university/teaches[IID = $i/IID]
              return $c/course_id }
        </instructor>
}
</university-1>
```

Figure 30.14 Creating nested structures in XQuery.

of values returned by a path expression may have some values repeated because they are repeated in the document, although an XPath expression result can contain at most one occurrence of each node in the document. XQuery supports many other functions; see the references in the bibliographical notes for more information. These functions are actually common to XPath 2.0 and XQuery and can be used in any XPath path expression.

To avoid namespace conflicts, functions are associated with a namespace:

http://www.w3.org/2005/xpath-functions

which has a default namespace prefix of fn. Thus, these functions can be referred to unambiguously as fn:sum or fn:count.

While XQuery does not provide a **group by** construct, aggregate queries can be written by using the aggregate functions on path or FLWOR expressions nested within the **return** clause. For example, the following query on the university XML schema finds the total salary of all instructors in each department:

```
for $d in /university/department
return
    <department-total-salary>
        <dept_name> { $d/dept_name } </dept_name>
        <total_salary> { fn:sum(
            for $i in /university/instructor[dept_name = $d/dept_name]
            return $i/salary
        ) } </total_salary>
    </department-total-salary>
```

30.4.3.4 Sorting of Results

Results can be sorted in XQuery by using the **order by** clause. For instance, this query outputs all instructor elements sorted by the name subelement:

```
for $i in /university/instructor
order by $i/name
return <instructor> { $i/* } </instructor>
```

To sort in descending order, we can use **order by** $i/name **descending**.

Sorting can be done at multiple levels of nesting. For instance, we can get a nested representation of university information with departments sorted in department name order, with courses sorted by course identifiers, as follows:

```
<university-1> {
    for $d in /university/department
    order by $d/dept_name
    return
        <department>
            { $d/* }
            { for $c in /university/course[dept_name = $d/dept_name]
            order by $c/course_id
            return <course> { $c/* } </course> }
        </department>
} </university-1>
```

30.4.3.5 Functions and Types

XQuery provides a variety of built-in functions, such as numeric functions and string matching and manipulation functions. In addition, XQuery supports user-defined functions. The following user-defined function takes as input an instructor identifier and returns a list of all courses offered by the department to which the instructor belongs:

```
declare function local:dept_courses($iid as xs:string) as element(course)* {
    for $i in /university/instructor[IID = $iid],
        $c in /university/courses[dept_name = $i/dept_name]
    return $c
}
```

The namespace prefix xs: used in the above example is predefined by XQuery to be associated with the XML Schema namespace, while the namespace local: is predefined to be associated with XQuery local functions.

The type specifications for function arguments and return values are optional, and may be omitted. XQuery uses the type system of XML Schema. The type element allows elements with any tag, while element(course) allows elements with the tag course. Types can be suffixed with a * to indicate a sequence of values of that type; for example, the definition of function dept_courses specifies the return value as a sequence of course elements.

The following query, which illustrates function invocation, prints out the department courses for the instructor(s) named Srinivasan:

```
for $i in /university/instructor[name = "Srinivasan"],
return local:inst_dept_courses($i/IID)
```

XQuery performs type conversion automatically whenever required. For example, if a numeric value represented by a string is compared to a numeric type, type conversion from string to the numeric type is done automatically. When an element is passed to a function that expects a string value, type conversion to a string is done by concatenating all the text values contained (nested) within the element. Thus, the function contains(a,b), which checks if string a contains string b, can be used with its first argument set to an element, in which case it checks if the element a contains the string b nested anywhere inside it. XQuery also provides functions to convert between types. For instance, number(x) converts a string to a number.

30.4.3.6 Other Features

XQuery offers a variety of other features, such as if-then-else constructs that can be used within **return** clauses, and existential and universal quantification that can be used in predicates in **where** clauses. For example, existential quantification can be expressed in the **where** clause by using:

```
some $e in path satisfies P
```

where path is a path expression and P is a predicate that can use $e. Universal quantification can be expressed by using **every** in place of **some**.

For example, to find departments where every instructor has a salary greater than $50,000, we can use the following query:

```
for $d in /university/department
where every $i in /university/instructor[dept_name=$d/dept_name]
        satisfies $i/salary > 50000
return $d
```

Note, however, that if a department has no instructor, it will trivially satisfy the above condition. An extra clause:

```
and fn:exists(/university/instructor[dept_name=$d/dept_name])
```

can be used to ensure that there is at least one instructor in the department. The built-in function exists() used in the clause returns true if its input argument is nonempty.

The XQJ standard provides an API to submit XQuery queries to an XML database system and to retrieve the XML results. Its functionality is similar to the JDBC API.

30.5 Application Program Interfaces to XML

With the wide acceptance of XML as a data representation and exchange format, software tools are widely available for manipulation of XML data. There are two standard models for programmatic manipulation of XML, each available for use with a number of popular programming languages. Both these APIs can be used to parse an XML document and create an in-memory representation of the document. They are used for applications that deal with individual XML documents. Note, however, that they are not suitable for querying large collections of XML data; declarative querying mechanisms such as XPath and XQuery are better suited to this task.

One of the standard APIs for manipulating XML is based on the *document object model* (DOM), which treats XML content as a tree, with each element represented by a node, called a DOMNode. Programs may access parts of the document in a navigational fashion, beginning with the root.

DOM libraries are available for most common programming languages and are even present in web browsers, where they may be used to manipulate the document displayed to the user. We outline here some of the interfaces and methods in the Java API for DOM, to give a flavor of DOM.

- The Java DOM API provides an interface called Node, and interfaces Element and Attribute, which inherit from the Node interface.
- The Node interface provides methods such as getParentNode(), getFirstChild(), and getNextSibling(), to navigate the DOM tree, starting with the root node.
- Subelements of an element can be accessed by name, using getElementsByTagName(name), which returns a list of all child elements with a specified tag name; individual members of the list can be accessed by the method item(i), which returns the *i*th element in the list.

- Attribute values of an element can be accessed by name, using the method getAttribute(name).
- The text value of an element is modeled as a Text node, which is a child of the element node; an element node with no subelements has only one such child node. The method getData() on the Text node returns the text contents.

DOM also provides a variety of functions for updating the document by adding and deleting attribute and element children of a node, setting node values, and so on.

Many more details are required for writing an actual DOM program; see the bibliographical notes for references to further information.

DOM can be used to access XML data stored in databases, and an XML database can be built with DOM as its primary interface for accessing and modifying data. However, the DOM interface does not support any form of declarative querying.

The second commonly used programming interface, the *Simple API for XML* (SAX), is an *event* model designed to provide a common interface between parsers and applications. This API is built on the notion of *event handlers*, which consist of user-specified functions associated with parsing events. Parsing events correspond to the recognition of parts of a document; for example, an event is generated when the start-tag is found for an element, and another event is generated when the end-tag is found. The pieces of a document are always encountered in order from start to finish.

The SAX application developer creates handler functions for each event and registers them. When a document is read in by the SAX parser, as each event occurs, the handler function is called with parameters describing the event (such as element tag or text contents). The handler functions then carry out their task. For example, to construct a tree representing the XML data, the handler functions for an attribute or element start event could add a node (or nodes) to a partially constructed tree. The start- and end-tag event handlers would also have to keep track of the current node in the tree to which new nodes must be attached; the element start event would set the new element as the node that is the point where further child nodes must be attached. The corresponding element end event would set the parent of the node as the current node where further child nodes must be attached.

SAX generally requires more programming effort than DOM, but it helps avoid the overhead of creating a DOM tree in situations where the application needs to create its own data representation. If DOM were used for such applications, there would be unnecessary space and time overhead for constructing the DOM tree.

30.6 Storage of XML Data

Many applications require storage of XML data. One way to store XML data are to store them as documents in a file system, while a second is to build a special-purpose database to store XML data. Another approach is to convert the XML data to a rela-

tional representation and store them in a relational database. Several alternatives for storing XML data are briefly outlined in this section.

30.6.1 Non-relational Data Stores

There are several alternatives for storing XML data in non-relational data-storage systems:

- **Store in flat files.** Since XML is primarily a file format, a natural storage mechanism is simply a flat file. This approach has many of the drawbacks, outlined in Chapter 1, of using file systems as the basis for database applications. In particular, it lacks data isolation, atomicity, concurrent access, and security. However, the wide availability of XML tools that work on file data makes it relatively easy to access and query XML data stored in files. Thus, this storage format may be sufficient for some applications.
- **Create an XML database.** XML databases are databases that use XML as their basic data model. Early XML databases implemented the Document Object Model on a C++-based object-oriented database. This allows much of the object-oriented database infrastructure to be reused, while providing a standard XML interface. The addition of XQuery or other XML query languages provides declarative querying. Other implementations have built the entire XML storage and querying infrastructure on top of a storage manager that provides transactional support.

Although several databases designed specifically to store XML data have been built, building a full-featured database system from the ground up is a very complex task. Such a database must support not only XML data storage and querying but also other database features such as transactions, security, support for data access from clients, and a variety of administration facilities. It makes sense to instead use an existing database system to provide these facilities and implement XML data storage and querying either on top of the relational abstraction or as a layer parallel to the relational abstraction. We study these approaches in Section 30.6.2.

30.6.2 Relational Databases

Since relational databases are widely used in existing applications, there is a great benefit to be had in storing XML data in relational databases, so that the data can be accessed from existing applications.

Converting XML data to relational form is usually straightforward if the data were generated from a relational schema in the first place and XML is used merely as a data exchange format for relational data. However, there are many applications where the XML data are not generated from a relational schema, and translating the data to relational form for storage may not be straightforward. In particular, nested elements and elements that recur (corresponding to set-valued attributes) complicate storage of

XML data in relational format. Several alternative approaches are available, which we describe below.

30.6.2.1 Store as String

Small XML documents can be stored as string (**clob**) values in tuples in a relational database. Large XML documents with the top-level element having many children can be handled by storing each child element as a string in a separate tuple. For instance, the XML data in Figure 30.1 could be stored as a set of tuples in a relation *elements*(*data*), with the attribute *data* of each tuple storing one XML element (department, course, instructor, or teaches) in string form.

While this representation is easy to use, the database system does not know the schema of the stored elements. As a result, it is not possible to query the data directly. In fact, it is not even possible to implement simple selections such as finding all department elements, or finding the department element with department name "Comp. Sci.", without scanning all tuples of the relation and examining the string contents.

A partial solution to this problem is to store different types of elements in different relations and also store the values of some critical elements as attributes of the relation to enable indexing. For instance, in our example, the relations would be *department_elements, course_elements, instructor_elements*, and *teaches_elements*, each with an attribute *data*. Each relation may have extra attributes to store the values of some subelements, such as *dept_name*, *course_id*, or *name*. Thus, a query that requires department elements with a specified department name can be answered efficiently with this representation. Such an approach depends on type information about XML data, such as the DTD of the data.

Some database systems, such as Oracle, support **function indices**, which can help avoid replication of attributes between the XML string and relation attributes. Unlike normal indices, which are on attribute values, function indices can be built on the result of applying user-defined functions on tuples. For instance, a function index can be built on a user-defined function that returns the value of the dept_name subelement of the XML string in a tuple. The index can then be used in the same way as an index on a *dept_name* attribute.

The approaches have the drawback that a large part of the XML information is stored within strings. It is possible to store all the information in relations in one of several ways that we examine next.

30.6.2.2 Tree Representation

Arbitrary XML data can be modeled as a tree and stored using a relation

$$nodes(id, parent\_id, type, label, value)$$

Each element and attribute in the XML data are given a unique identifier. A tuple is inserted in the *nodes* relation for each element and attribute with its identifier (*id*), the

identifier of its parent node (*parent_id*), the type of the node (attribute or element), the name of the element or attribute (*label*), and the text value of the element or attribute (*value*).

If order information of elements and attributes must be preserved, an extra attribute *position* can be added to the *nodes* relation to indicate the relative position of the child among the children of the parent. As an exercise, you can represent the XML data of Figure 30.1 by using this technique.

This representation has the advantage that all XML information can be represented directly in relational form, and many XML queries can be translated into relational queries and executed inside the database system. However, it has the drawback that each element gets broken up into many pieces, and a large number of joins are required to reassemble subelements into an element.

30.6.2.3 Map to Relations

In this approach, XML elements whose schema is known are mapped to relations and attributes. Elements whose schema is unknown are stored as strings or as a tree.

A relation is created for each element type (including subelements) whose schema is known and whose type is a complex type (i.e., contains attributes or subelements). The root element of the document can be ignored in this step if it does not have any attributes. The attributes of the relation are defined as follows:

- All attributes of these elements are stored as string-valued attributes of the relation.
- If a subelement of the element is a simple type (i.e., cannot have attributes or subelements), an attribute is added to the relation to represent the subelement. The type of the relation attribute defaults to a string value, but if the subelement had an XML Schema type, a corresponding SQL type may be used.

  For example, when applied to the element *department* in the schema (DTD or XML Schema) of the data in Figure 30.1, the subelements *dept_name*, *building*, and *budget* of the element *department* all become attributes of a relation *department*. Applying this procedure to the remaining elements, we get back the original relational schema that we have used in earlier chapters.
- Otherwise, a relation is created corresponding to the subelement (using the same rules recursively on its subelements). Further:
  - An identifier attribute is added to the relations representing the element. (The identifier attribute is added only once even if an element has several subelements.)
  - An attribute *parent_id* is added to the relation representing the subelement, storing the identifier of its parent element.
  - If ordering is to be preserved, an attribute *position* is added to the relation representing the subelement.

For example, if we apply the above procedure to the schema corresponding to the data in Figure 30.5, we get the following relations:

department(*id*, *dept_name*, *building*, *budget*)
course(*parent_id*, *course_id*, *dept_name*, *title*, *credits*)

Variants of this approach are possible. For example, the relations corresponding to subelements that can occur at most once can be "flattened" into the parent relation by moving all their attributes into the parent relation. The bibliographical notes provide references to different approaches to represent XML data as relations.

30.6.2.4 Publishing and Shredding XML Data

When XML is used to exchange data between business applications, the data most often originate in relational databases. Data in relational databases must be *published*, that is, converted to XML form, for export to other applications. Incoming data must be *shredded*, that is, converted back from XML to normalized relation form and stored in a relational database. While application code can perform the publishing and shredding operations, the operations are so common that the conversions should be done automatically, without writing application code, where possible. Database vendors have spent a lot of effort to *XML-enable* their database products.

An XML-enabled database supports an automatic mechanism for publishing relational data as XML. The mapping used for publishing data may be simple or complex. A simple relation to XML mapping might create an XML element for every row of a table and make each column in that row a subelement of the XML element. The XML schema in Figure 30.1 can be created from a relational representation of university information, using such a mapping. Such a mapping is straightforward to generate automatically. Such an XML view of relational data can be treated as a *virtual* XML document, and XML queries can be executed against the virtual XML document.

A more complicated mapping would allow nested structures to be created. Extensions of SQL with nested queries in the **select** clause have been developed to allow easy creation of nested XML output. We outline these extensions in Section 30.6.3.

Mappings also have to be defined to shred XML data into a relational representation. For XML data created from a relational representation, the mapping required to shred the data are a straightforward inverse of the mapping used to publish the data. For the general case, a mapping can be generated as outlined in Section 30.6.2.3.

30.6.2.5 Native Storage within a Relational Database

Some relational databases support **native storage** of XML. Such systems store XML data as strings or in more efficient binary representations, without converting the data to relational form. A new data type **xml** is introduced to represent XML data, although the CLOB and BLOB data types may provide the underlying storage mechanism. XML query languages such as XPath and XQuery are supported to query XML data.

A relation with an attribute of type **xml** can be used to store a collection of XML documents; each document is stored as a value of type **xml** in a separate tuple. Special-purpose indices are created to index the XML data.

Several database systems provide native support for XML data. They provide an **xml** data type and allow XQuery queries to be embedded within SQL queries. An XQuery query can be executed on a single XML document and can be embedded within an SQL query to allow it to execute on each of a collection of documents, with each document stored in a separate tuple.

30.6.3 SQL/XML

While XML is used widely for data interchange, structured data are still widely stored in relational databases. There is often a need to convert relational data to XML representation. The SQL/XML standard, developed to meet this need, defines a standard extension of SQL, allowing the creation of nested XML output. The standard has several parts, including a standard way of mapping SQL types to XML Schema types, and a standard way to map relational schemas to XML schemas, as well as SQL query language extensions.

For example, the SQL/XML representation of the *department* relation would have an XML schema with outermost element *department*, with each tuple mapped to an XML element *row*, and each relation attribute mapped to an XML element of the same name (with some conventions to resolve incompatibilities with special characters in names). An entire SQL schema, with multiple relations, can also be mapped to XML in a similar fashion. Figure 30.15 shows the SQL/XML representation of (part of) the *university* data from Figure 30.1, containing the relations *department* and *course*.

SQL/XML adds several operators and aggregate operations to SQL to allow the construction of XML output directly from the extended SQL. The **xmlelement** function can be used to create XML elements, while **xmlattributes** can be used to create attributes, as illustrated by the following query.

```
select xmlelement (name "course",
                   xmlattributes (course_id as course_id, dept_name as dept_name),
                   xmlelement (name "title", title),
                   xmlelement (name "credits", credits))
        from course
```

This query creates an XML element for each course, with the course identifier and department name represented as attributes, and title and credits as subelements. The result would look like the course elements shown in Figure 30.11, but without the instructor attribute. The **xmlattributes** operator creates the XML attribute name using the SQL attribute name, which can be changed using an **as** clause as shown.

The **xmlforest** operator simplifies the construction of XML structures. Its syntax and behavior are similar to those of **xmlattributes**, except that it creates a forest (collection) of subelements, instead of a list of attributes. It takes multiple arguments, creating

```
<university>
    <department>
        <row>
            <dept_name> Comp. Sci. </dept_name>
            <building> Taylor </building>
            <budget> 100000 </budget>
        </row>
        <row>
            <dept_name> Biology </dept_name>
            <building> Watson </building>
            <budget> 90000 </budget>
        </row>
    </department>
    <course>
        <row>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <dept_name> Comp. Sci </dept_name>
            <credits> 4 </credits>
        </row>
        <row>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <dept_name> Biology </dept_name>
            <credits> 4 </credits>
        </row>
    </course>
</university>
```

Figure 30.15 SQL/XML representation of (part of) university information.

an element for each argument, with the attribute's SQL name used as the XML element name. The **xmlconcat** operator can be used to concatenate elements created by subexpressions into a forest.

When the SQL value used to construct an attribute is null, the attribute is omitted. Null values are omitted when the body of an element is constructed.

SQL/XML also provides an aggregate function **xmlagg** that creates a forest (collection) of XML elements from the collection of values on which it is applied. The following query creates an element for each department with a course, containing as subelements all the courses in that department. Since the query has a clause **group by** *dept_name*, the aggregate function is applied on all courses in each department, creating a sequence of *course_id* elements.

```
select xmlelement (name "department",
                   dept_name,
                   xmlagg (xmlforest(course_id)
                           order by course_id))
from course
group by dept_name
```

SQL/XML allows the sequence created by **xmlagg** to be ordered, as illustrated in the preceding query. See the bibliographical notes for references to more information on SQL/XML.

30.7 XML Applications

We now outline several applications of XML for storing and communicating (exchanging) data and for accessing web services (information resources).

30.7.1 Storing Data with Complex Structure

Many applications need to store data that are structured, but are not easily modeled as relations. Consider, such as, user preferences that must be stored by an application such as a browser. There are usually a large number of fields, such as home page, security settings, language settings, and display settings, that must be recorded. Some of the fields are multivalued, for example, a list of trusted sites, or maybe ordered lists, for example, a list of bookmarks. Applications traditionally used some type of textual representation to store such data. Today, a majority of such applications prefer to store such configuration information in XML format. The ad hoc textual representations used earlier require effort to design and effort to create parsers that can read the file and convert the data into a form that a program can use. The XML representation avoids both these steps.

XML-based representations are now widely used for storing documents, spreadsheet data, and other data that are part of office application packages. The *Open Document Format* (*ODF*), supported by the Open Office software suite as well as other office suites, and the *Office Open XML* (*OOXML*) format, supported by the Microsoft Office suite, are document representation standards based on XML. They are the two most widely used formats for editable document representation.

XML is also used to represent data with complex structure that must be exchanged between different parts of an application. For example, a database system may represent a query execution plan (a relational-algebra expression with extra information on how to execute operations) by using XML. This allows one part of the system to generate the query execution plan and another part to display it, without using a shared data structure. For example, the data may be generated at a server system and sent to a client system where the data are displayed.

30.7.2 Standardized Data Exchange Formats

XML-based standards for representation of data have been developed for a variety of specialized applications, ranging from business applications such as banking and shipping to scientific applications such as chemistry and molecular biology. Some examples:

- The chemical industry needs information about chemicals, such as their molecular structure, and a variety of important properties, such as boiling and melting points, calorific values, and solubility in various solvents. *ChemML* is a standard for representing such information.
- In shipping, carriers of goods and customs and tax officials need shipment records containing detailed information about the goods being shipped, from whom and to where they were sent, to whom and to where they are being shipped, the monetary value of the goods, and so on.
- An online marketplace in which business can buy and sell goods [a so-called business-to-business (B2B) market] requires information such as product catalogs, including detailed product descriptions and price information, product inventories, quotes for a proposed sale, and purchase orders. For example, the *RosettaNet* standards for e-business applications define XML schemas and semantics for representing data as well as standards for message exchange.

Using normalized relational schemas to model such complex data requirements would result in a large number of relations that do not correspond directly to the objects that are being modeled. The relations would often have large numbers of attributes; explicit representation of attribute/element names along with values in XML helps avoid confusion between attributes. Nested element representations help reduce the number of relations that must be represented, as well as the number of joins required to get required information, at the possible cost of redundancy. For instance, in our university example, listing departments with course elements nested within department elements, as in Figure 30.5, results in a format that is more natural for some applications—in particular, for humans to read—than is the normalized representation in Figure 30.1.

30.7.3 Web Services

Applications often require data from outside of the organization, or from another department in the same organization that uses a different database. In many such situations, the outside organization or department is not willing to allow direct access to its database using SQL, but is willing to provide limited forms of information through predefined interfaces.

When the information is to be used directly by a human, organizations provide web-based forms, where users can input values and get back desired information in

HTML form. However, there are many applications where such information needs to be accessed by software programs rather than by end users. Providing the results of a query in XML form is a clear requirement. In addition, it makes sense to specify the input values to the query also in XML format.

In effect, the provider of the information defines procedures whose input and output are both in XML format. The HTTP protocol is used to communicate the input and output information, since it is widely used and can go through firewalls that institutions use to keep out unwanted traffic from the Internet.

The **Simple Object Access Protocol** (**SOAP**) defines a standard for invoking procedures, using XML for representing the procedure input and output. SOAP defines a standard XML schema for representing the name of the procedure and result status indicators such as failure/error indicators. The procedure parameters and results are application-dependent XML data embedded within the SOAP XML headers.

Typically, HTTP is used as the transport protocol for SOAP, but a message-based protocol (such as email over the SMTP protocol) may also be used. The SOAP standard is widely used today. For example, Amazon and Google provide SOAP-based procedures to carry out search and other activities. These procedures can be invoked by other applications that provide higher-level services to users. The SOAP standard is independent of the underlying programming language, and it is possible for a site running one language, such as C#, to invoke a service that runs on a different language, such as Java.

A site providing such a collection of SOAP procedures is called a **web service**. Several standards have been defined to support web services. The **Web Services Description Language** (**WSDL**) is a language used to describe a web service's capabilities. WSDL provides facilities that interface definitions (or function definitions) provide in a traditional programming language, specifying what functions are available and their input and output types. In addition, WSDL allows specification of the URL and network port number to be used to invoke the web service. There is also a standard called **Universal Description, Discovery, and Integration** (**UDDI**) that defines how a directory of available web services may be created and how a program may search in the directory to find a web service satisfying its requirements.

The following example illustrates the value of web services. An airline may define a web service providing a set of procedures that can be invoked by a travel web site; these may include procedures to find flight schedules and pricing information, as well as to make flight bookings. The travel web site may interact with multiple web services, provided by different airlines, hotels, and other companies, to provide travel information to a customer and to make travel bookings. By supporting web services, the individual companies allow a useful service to be constructed on top, integrating the individual services. Users can interact with a single web site to make their travel bookings without having to contact multiple separate web sites.

To invoke a web service, a client must prepare an appropriate SOAP XML message and send it to the service; when it gets the result encoded in XML, the client must then

extract information from the XML result. There are standard APIs in languages such as Java and C# to create and extract information from SOAP messages.

See the bibliographical notes for references to more information on web services.

30.7.4 Data Mediation

Comparison shopping is an example of a mediation application, in which data about items, inventory, pricing, and shipping costs are extracted from a variety of web sites offering a particular item for sale. The resulting aggregated information is significantly more valuable than the individual information offered by a single site.

A personal financial manager is a similar application in the context of banking. Consider a consumer with a variety of accounts to manage, such as bank accounts, credit-card accounts, and retirement accounts. Suppose that these accounts may be held at different institutions. Providing centralized management for all accounts of a customer is a major challenge. XML-based mediation addresses the problem by extracting an XML representation of account information from the respective web sites of the financial institutions where the individual holds accounts. This information may be extracted easily if the institution exports it in a standard XML format, for example, as a web service. For those that do not, *wrapper* software is used to generate XML data from HTML web pages returned by the web site. Wrapper applications need constant maintenance since they depend on formatting details of web pages, which change often. Nevertheless, the value provided by mediation often justifies the effort required to develop and maintain wrappers.

Once the basic tools are available to extract information from each source, a *mediator* application is used to combine the extracted information under a single schema. This may require further transformation of the XML data from each site, since different sites may structure the same information differently. They may also use different names for the same information (for instance, acct_number and account_id) or may even use the same name for different information. The mediator must decide on a single schema that represents all required information and must provide code to transform data between different representations. XML query languages such as XSLT and XQuery play an important role in the task of transformation between different XML representations.

30.8 Summary

- Like the Hyper-Text Markup Language (HTML) on which the web is based, the Extensible Markup Language (XML) is derived from the Standard Generalized Markup Language (SGML). XML was originally intended for providing functional markup for web documents, but it has now become the de facto standard data format for data exchange between applications.
- XML documents contain elements with matching starting and ending tags indicating the beginning and end of an element. Elements may have subelements nested

within them, to any level of nesting. Elements may also have attributes. The choice between representing information as attributes and subelements is often arbitrary in the context of data representation.

- Elements may have an attribute of type ID that stores a unique identifier for the element. Elements may also store references to other elements by using attributes of type IDREF. Attributes of type IDREFS can store a list of references.
- Documents optionally may have their schema specified by a document type definition (DTD). The DTD of a document specifies what elements may occur, how they may be nested, and what attributes each element may have. Although DTDs are widely used, they have several limitations. For instance, they do not provide a type system.
- XML Schema is now the standard mechanism for specifying the schema of an XML document. It provides a large set of basic types, as well as constructs for creating complex types and specifying integrity constraints, including key constraints and foreign-key (keyref) constraints.
- XML data can be represented as tree structures, with nodes corresponding to elements and attributes. Nesting of elements is reflected by the parent-child structure of the tree representation.
- Path expressions can be used to traverse the XML tree structure and locate data. XPath is a standard language for path expressions. It allows required elements to be specified by a file-system-like path, and it also allows selections and other features. XPath also forms part of other XML query languages.
- The XQuery language is the standard language for querying XML data. It has a structure not unlike SQL, with **for**, **let**, **where**, **order by**, and **return** clauses. However, it supports many extensions to deal with the tree nature of XML and to allow for the transformation of XML documents into other documents with a significantly different structure. XPath path expressions form a part of XQuery. XQuery supports nested queries and user-defined functions.
- The DOM and SAX APIs are widely used for programmatic access to XML data. These APIs are available from a variety of programming languages.
- XML data can be stored in any of several different ways. XML data may also be stored in file systems, or in XML databases, which use XML as their internal representation.
- XML data can be stored as strings in a relational database. Alternatively, relations can represent XML data as trees. As another alternative, XML data can be mapped to relations in the same way that E-R schemas are mapped to relational schemas. Native storage of XML in relational databases is facilitated by adding an **xml** data type to SQL.

- XML is used in a variety of applications, such as storing complex data, exchange of data between organizations in a standardized form, data mediation, and web services. Web services provide a remote-procedure call interface, with XML as the mechanism for encoding parameters as well as results.

Review Terms

- Extensible Markup Language (XML)
- Hyper-Text Markup Language (HTML)
- Standard Generalized Markup Language
- Markup language
- Tags
- Self-documenting
- Element
- Root element
- Nested elements
- Attribute
- Namespace
- Default namespace
- Schema definition
- Document Type Definition (DTD)
  - ID
  - IDREF and IDREFS
- XML Schema
  - Simple and complex types
  - Sequence type
  - Key and keyref
  - Occurrence constraints
- Tree model of XML data
- Nodes
- Querying and transformation
- Path expressions
- XPath
- XQuery
  - FLWOR expressions
    - **for**
    - **let**
    - **where**
    - **order by**
    - **return**
  - Joins
  - Nested FLWOR expression
  - Sorting
- XML API
- Document Object Model (DOM)
- Simple API for XML (SAX)
- Storage of XML data
  - In non-relational data stores
  - In relational databases
    - Store as string
    - Tree representation
    - Map to relations
    - Publish and shred
    - XML-enabled database
    - Native storage
    - SQL/XML
- XML applications

- Storing complex data
- Exchange of data
- Data mediation
- SOAP
- Web services

Practice Exercises

30.1 Write a query in XQuery on the XML representation in Figure 30.11 to find the total salary of all instructors in each department.

30.2 Write a query in XQuery on the XML representation in Figure 30.1 to compute the left outer join of department elements with course elements. (Hint: Use universal quantification.)

30.3 Write queries in XQuery to output course elements with associated instructor elements nested within the course elements, given the university information representation using ID and IDREFS in Figure 30.11.

30.4 Give a relational schema to represent bibliographical information specified according to the DTD fragment shown below:

```
<!DOCTYPE bibliography [
    <!ELEMENT book (title, author+, year, publisher, place?)>
    <!ELEMENT article (title, author+, journal, year, number, volume, pages?)>
    <!ELEMENT author ( last_name, first_name) >
    <!ELEMENT title ( #PCDATA )>
    ··· similar PCDATA declarations for year, publisher, place, journal, year,
        number, volume, pages, last_name and first_name
] >
```

The relational schema must keep track of the order of author elements. You can assume that only books and articles appear as top-level elements in XML documents.

30.5 Show the tree representation of the XML data in Figure 30.1, and the representation of the tree using *nodes* and *child* relations described in Section 30.6.2.

30.6 Consider the following recursive DTD:

```
<!DOCTYPE parts [
    <!ELEMENT part (name, subpartinfo*)>
    <!ELEMENT subpartinfo (part, quantity)>
    <!ELEMENT name ( #PCDATA )>
    <!ELEMENT quantity ( #PCDATA )>
] >
```

a. Show how to map this DTD to a relational schema. You can assume that part names are unique; that is, wherever a part appears, its subpart structure will be the same.

b. Create a schema in XML Schema corresponding to this DTD.

Exercises

30.7 Show, by giving a DTD, how to represent the non-1NF *books* relation from Section 29.1, using XML.

30.8 Consider the schema:

Emp = (ename, ChildrenSet **setof***(Children), SkillsSet* **setof***(Skills))*
Children = (name, Birthday)
Birthday = (day, month, year)
Skills = (type, ExamsSet **setof***(Exams))*
Exams = (year, city)

Write the following queries in XQuery:

a. Find the names of all employees who have a child who has a birthday in March.

b. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

c. List all skill types in *Emp*.

30.9 One way to share an XML document is to use XQuery to convert the schema to an SQL/XML mapping of the corresponding relational schema, and then use the SQL/XML mapping in the backward direction to populate the relation.

As an illustration, give an XQuery query to convert data from the *university-1* XML schema to the SQL/XML schema shown in Figure 30.15.

30.10 Consider the XML data shown in Figure 30.3. Suppose we wish to find purchase orders that ordered two or more copies of the part with identifier 123. Consider the following attempt to solve this problem:

```
for $p in purchaseorder
where $p/part/id = 123 and $p/part/quantity >= 2
return $p
```

Explain why the query may return some purchase orders that order less than two copies of part 123. Give a correct version of the above query.

30.11 Give a query in XQuery to flip the nesting of data from Exercise 30.7. That is, at the outermost level of nesting, the output must have elements corresponding to authors, and each such element must have nested within it items corresponding to all the books written by the author.

30.12 Consider the bibliography DTD fragment:

```
<!DOCTYPE bibliography [
    <!ELEMENT book (title, author+, year, publisher, place?)>
    <!ELEMENT article (title, author+, journal, year, number, volume, pages?)>
    <!ELEMENT author ( last_name, first_name) >
    <!ELEMENT title ( #PCDATA )>
    ··· similar PCDATA declarations for year, publisher, place, journal, year,
        number, volume, pages, last_name and first_name
] >
```

Write the following queries in XQuery:

a. Find all authors who have authored a book and an article in the same year.

b. Display books and articles sorted by year.

c. Display books with more than one author.

d. Find all books that contain the word *database* in their title and the word *Hank* in an author's name (whether first or last).

30.13 Give a relational mapping of the XML purchase order schema illustrated in Figure 30.3, using the approach described in Section 30.6.2.3. Suggest how to remove redundancy in the relational schema if item identifiers functionally determine the description and purchase and supplier names functionally determine the purchase and supplier address, respectively.

30.14 Write queries in SQL/XML to convert university data from the relational schema we have used in earlier chapters to the *university-1* and *university-2* XML schemas.

30.15 Consider the example XML schema from Section 30.3.2, and write XQuery queries to carry out the following tasks:

a. Check if the key constraint shown in Section 30.3.2 holds.

b. Check if the keyref constraint shown in Section 30.3.2 holds.

30.16 Consider Exercise 30.4, and suppose that authors could also appear as top-level elements. What change would have to be done to the relational schema?

30.17 As in Exercise 30.15, write queries to convert university data to the *university-1* and *university-2* XML schemas, but this time by writing XQuery queries on the default SQL/XML database to XML mapping.

Tools

A number of tools to deal with XML are available in the public domain. The W3C web site www.w3.org has pages describing the various XML-related standards, as well as pointers to software tools such as language implementations. An extensive list of XQuery implementations is available at www.w3.org/XML/Query. Saxon D (saxon.sourceforge.net) and Galax (www.galaxquery.org/) are useful as learning tools, although not designed to handle large databases. Exist (exist-db.org) is an open-source XML database, supporting a variety of features. Several commercial databases, including IBM DB2, Oracle, and Microsoft SQL Server, support XML storage, publishing using various SQL extensions, and querying using XPath and XQuery.

Further Reading

The World Wide Web Consortium (W3C) acts as the standards body for web-related standards, including basic XML and all the XML-related languages such as XPath, XSLT, and XQuery. A large number of technical reports defining the XML-related standards are available at www.w3.org. This site also contains tutorials and pointers to software implementing the various standards.

The XQuery language derives from an XML query language called Quilt; Quilt itself included features from earlier languages such as XPath, discussed in Section 30.4.2, and two other XML query languages, XQL and XML-QL. Quilt is described in [Chamberlin et al. (2000)].

Bibliography

[Chamberlin et al. (2000)] D. D. Chamberlin, J. Robie, and D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources", In *Proc. of the International Workshop on the Web and Databases (WebDB)* (2000), pages 53–62.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.