chapter 13

XML: Extensible Markup Language

Many Internet applications provide Web interfaces to access information stored in one or more databases. These databases are often referred to as **data sources**. It is common to use the three-tier client/server architectures for Internet applications (see Section 2.5). Internet database applications are designed to interact with the user through Web interfaces that display Web pages on desktops, laptops, and mobile devices. The common method of specifying the contents and formatting of Web pages is through the use of **hypertext documents**. There are various languages for writing these documents, the most common being HTML (HyperText Markup Language). Although HTML is widely used for formatting and structuring Web *documents*, it is not suitable for specifying *structured data* that is extracted from databases. A new language—namely, XML (Extensible Markup Language)—has emerged as the standard for structuring and exchanging data over the Web in text files. Another language that can be used for the same purpose is JSON (JavaScript Object Notation; see Section 11.4). XML can be used to provide information about the structure and meaning of the data in the Web pages rather than just specifying how the Web pages are formatted for display on the screen. Both XML and JSON documents provide descriptive information, such as attribute names, as well as the values of these attributes, in a text file; hence, they are known as **self-describing documents**. The formatting aspects of Web pages are specified separately—for example, by using a formatting language such as XSL (Extensible Stylesheet Language) or a transformation language such as XSLT (Extensible Stylesheet Language for Transformations or simply XSL Transformations). Recently, XML has also been proposed as a possible model for data storage and retrieval, although only a few experimental database systems based on XML have been developed so far.

Basic HTML is useful for generating *static* Web pages with fixed text and other objects, but most e-commerce applications require Web pages that provide interactive features with the user and use the information provided by the user for selecting specific data from a database for display. Such Web pages are called *dynamic* Web pages, because the data extracted and displayed each time will be different depending on user input. For example, a banking app would get the user's account number, then extract the balance for that user's account from the database for display. We discussed how scripting languages, such as PHP, can be used to generate dynamic Web pages for applications such as those presented in Chapter 11. XML can be used to transfer information in self-describing textual files among various programs on different computers when needed by the applications.

In this chapter, we will focus on describing the XML data model and its associated languages, and how data extracted from relational databases can be formatted as XML documents to be exchanged over the Web. Section 13.1 discusses the difference among structured, semistructured, and unstructured data. Section 13.2 presents the XML data model, which is based on tree (hierarchical) structures as compared to the flat relational data model structures. In Section 13.3, we focus on the structure of XML documents and the languages for specifying the structure of these documents, such as DTD (Document Type Definition) and XML Schema. Section 13.4 shows the relationship between XML and relational databases. Section 13.5 describes some of the languages associated with XML, such as XPath and XQuery. Section 13.6 discusses how data extracted from relational databases can be formatted as XML documents. In Section 13.7, we discuss the new functions that have been incorporated into XML for the purpose of generating XML documents from relational databases. Finally, Section 13.8 is the chapter summary.

13.1 Structured, Semistructured, and Unstructured Data

The information stored in relational databases is known as **structured data** because it is represented in a strict format. For example, each record in a relational database table—such as each of the tables in the COMPANY database in Figure 5.6—follows the same format as the other records. For structured data, it is common to carefully design the database schema using techniques such as those described in Chapters 3 and 4 in order to define the database structure. The DBMS then checks to ensure that all data follows the structures and constraints specified in the schema.

However, not all data is collected and inserted into carefully designed structured databases. In some applications, data is collected in an ad hoc manner before it is known how it will be stored and managed. This data may have a certain structure, but not all the information collected will have the identical structure. Some attributes may be shared among the various entities, but other attributes may exist only in a few entities. Moreover, additional attributes can be introduced in some of the newer data items at any time, and there is no predefined schema. This type of data is known as **semistructured data**. A number of data models have been introduced

for representing semistructured data, often based on using tree or graph data structures rather than the flat relational model structures.

A key difference between structured and semistructured data concerns how the schema constructs (such as the names of attributes, relationships, and entity types) are handled. In semistructured data, the schema information is *mixed in* with the data values, since each data object can have different attributes that are not known in advance. Hence, this type of data is sometimes referred to as **self-describing data**. Many of the newer NOSQL systems adopt self-describing storage schemes (see Chapter 24). Consider the following example. We want to collect a list of bibliographic references related to a certain research project. Some of these may be books or technical reports, others may be research articles in journals or conference proceedings, and still others may refer to complete journal issues or conference proceedings. Clearly, each of these may have different attributes and different types of information. Even for the same type of reference—say, conference articles—we may have different information. For example, one article citation may be complete, with full information about author names, title, proceedings, page numbers, and so on, whereas another citation may not have all the information available. New types of bibliographic sources may appear in the future—for instance, references to Web pages or to conference tutorials—and these may have new attributes that describe them.

One model for displaying semistructured data is a directed graph, as shown in Figure 13.1. The information shown in Figure 13.1 corresponds to some of the structured data shown in Figure 5.6. As we can see, this model somewhat resembles the object model (see Section 12.1.3) in its ability to represent complex objects and nested structures. In Figure 13.1, the **labels** or **tags** on the directed edges represent the schema names: the *names of attributes, object types* (or *entity types*

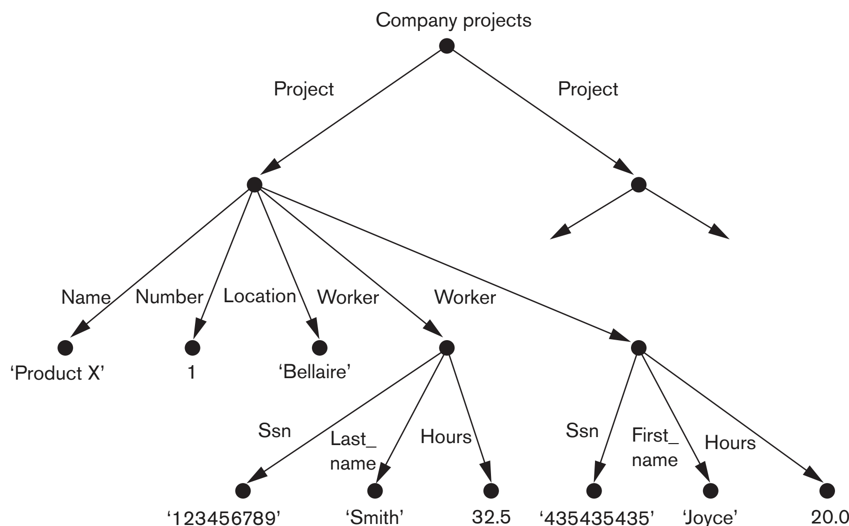

Figure 13.1
Representing semistructured data as a graph.

or *classes*), and *relationships*. The internal nodes represent individual objects or composite attributes. The leaf nodes represent actual data values of simple (atomic) attributes.

There are two main differences between the semistructured model and the object model that we discussed in Chapter 12:

1. The schema information—names of attributes, relationships, and classes (object types) in the semistructured model—is intermixed with the objects and their data values in the same data structure.
2. In the semistructured model, there is no requirement for a predefined schema to which the data objects must conform, although it is possible to define a schema if necessary. The object model of Chapter 12 requires a schema.

In addition to structured and semistructured data, a third category exists, known as **unstructured data** because there is very limited indication of the type of data. A typical example is a text document that contains information embedded within it. Web pages in HTML that contain some data are considered to be unstructured data. Consider part of an HTML file, shown in Figure 13.2. Text that appears between angled brackets, <…>, is an **HTML tag**. A tag with a slash, </…>, indicates an **end tag**, which represents the ending of the effect of a matching **start tag**. The tags **mark up** the document[1] in order to instruct an HTML processor how to display the text between a start tag and a matching end tag. Hence, the tags specify document formatting rather than the meaning of the various data elements in the document. HTML tags specify information, such as font size and style (boldface, italics, and so on), color, heading levels in documents, and so on. Some tags provide text structuring in documents, such as specifying a numbered or unnumbered list or a table. Even these structuring tags specify that the embedded textual data is to be displayed in a certain manner rather than indicating the type of data represented in the table.

HTML uses a large number of predefined tags, which are used to specify a variety of commands for formatting Web documents for display. The start and end tags specify the range of text to be formatted by each command. A few examples of the tags shown in Figure 13.2 follow:

- The <HTML> … </HTML> tags specify the boundaries of the document.
- The **document header** information—within the <HEAD> … </HEAD> tags—specifies various commands that will be used elsewhere in the document. For example, it may specify various **script functions** in a language such as JavaScript or PERL, or certain **formatting styles** (fonts, paragraph styles, header styles, and so on) that can be used in the document. It can also specify a title to indicate what the HTML file is for, and other similar information that will not be displayed as part of the document.

---

[1]That is why it is known as HyperText *Markup* Language.

```
<HTML>
    <HEAD>
    …
    </HEAD>
    <BODY>
        <H1>List of company projects and the employees in each project</H1>
        <H2>The ProductX project:</H2>
        <TABLE width="100%" border=0 cellpadding=0 cellspacing=0>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">John Smith:</FONT></TD>
                <TD>32.5 hours per week</TD>
            </TR>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">Joyce English:</FONT></TD>
                <TD>20.0 hours per week</TD>
            </TR>
        </TABLE>
        <H2>The ProductY project:</H2>
        <TABLE width="100%" border=0 cellpadding=0 cellspacing=0>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">John Smith:</FONT></TD>
                <TD>7.5 hours per week</TD>
            </TR>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">Joyce English:</FONT></TD>
                <TD>20.0 hours per week</TD>
            </TR>
            <TR>
                <TD width="50%"><FONT size="2" face="Arial">Franklin Wong:</FONT></TD>
                <TD>10.0 hours per week</TD>
            </TR>
        </TABLE>
    …
    </BODY>
</HTML>
```

Figure 13.2
Part of an HTML document representing unstructured data.

- The **body** of the document—specified within the <BODY> … </BODY> tags—includes the document text and the markup tags that specify how the text is to be formatted and displayed. It can also include references to other objects, such as images, videos, voice messages, and other documents.
- The <H1> … </H1> tags specify that the text is to be displayed as a level 1 heading. There are many heading levels (<H2>, <H3>, and so on), each displaying text in a less prominent heading format.
- The <TABLE> … </TABLE> tags specify that the following text is to be displayed as a table. Each *table row* in the table is enclosed within <TR> … </TR>

tags, and the individual table data elements in a row are displayed within <TD> … </TD> tags.[2]

- Some tags may have **attributes**, which appear within the start tag and describe additional properties of the tag.[3]

In Figure 13.2, the <TABLE> start tag has four attributes describing various characteristics of the table. The following <TD> and <FONT> start tags have one and two attributes, respectively.

HTML has a very large number of predefined tags, and whole books are devoted to describing how to use these tags. If designed properly, HTML documents can be formatted so that humans are able to easily understand the document contents and are able to navigate through the resulting Web documents. However, the source HTML text documents are very difficult to interpret automatically by *computer programs* because they do not include schema information about the type of data in the documents. As e-commerce and other Internet applications become increasingly automated, it is becoming crucial to be able to exchange Web documents among various computer sites and to interpret their contents automatically. This need was one of the reasons that led to the development of XML. In addition, an extendible version of HTML called XHTML was developed that allows users to extend the tags of HTML for different applications and allows an XHTML file to be interpreted by standard XML processing programs. Our discussion will focus on XML only.

The example in Figure 13.2 illustrates a **static** HTML page, since all the information to be displayed is explicitly spelled out as fixed text in the HTML file. In many cases, some of the information to be displayed may be extracted from a database. For example, the project names and the employees working on each project may be extracted from the database in Figure 5.6 through the appropriate SQL query. We may want to use the same HTML formatting tags for displaying each project and the employees who work on it, but we may want to change the particular projects (and employees) being displayed. For example, we may want to see a Web page displaying the information for *ProjectX*, and then later a page displaying the information for *ProjectY*. Although both pages are displayed using the same HTML formatting tags, the actual data items displayed will be different. Such Web pages are called **dynamic**, since the data parts of the page may be different each time it is displayed, even though the display appearance is the same. We discussed in Chapter 11 how scripting languages, such as PHP, can be used to generate dynamic Web pages.

13.2 XML Hierarchical (Tree) Data Model

We now introduce the data model used in XML. The basic object in XML is the XML document. Two main structuring concepts are used to construct an XML document: **elements** and **attributes**. It is important to note that the term *attribute* in XML is *not*

[2]<TR> stands for table row and <TD> stands for table data.

[3]This is how the term *attribute* is used in document markup languages, which differs from how it is used in database models.

used in the same manner as is customary in database terminology, but rather as it is used in document description languages such as HTML and SGML.[4] Attributes in XML provide additional information that describes elements, as we will see. There are additional concepts in XML, such as entities, identifiers, and references, but first we concentrate on describing elements and attributes to show the essence of the XML model.

Figure 13.3 shows an example of an XML element called <Projects>. As in HTML, elements are identified in a document by their start tag and end tag. The tag names are enclosed between angled brackets < … >, and end tags are further identified by a slash, </ … >.[5]

Complex elements are constructed from other elements hierarchically, whereas **simple elements** contain data values. A major difference between XML and HTML is that XML tag names are defined to describe the meaning of the data elements in the document rather than to describe how the text is to be displayed. This makes it possible to process the data elements in the XML document automatically by computer programs. Also, the XML tag (element) names can be defined in another document, known as the *schema document*, to give a semantic meaning to the tag names that can be exchanged among multiple programs and users. In HTML, all tag names are predefined and fixed; that is why they are not extendible.

It is straightforward to see the correspondence between the XML textual representation shown in Figure 13.3 and the tree structure shown in Figure 13.1. In the tree representation, internal nodes represent complex elements, whereas leaf nodes represent simple elements. That is why the XML model is called a **tree model** or a **hierarchical model**. In Figure 13.3, the simple elements are the ones with the tag names <Name>, <Number>, <Location>, <Dept_no>, <Ssn>, <Last_name>, <First_name>, and <Hours>. The complex elements are the ones with the tag names <Projects>, <Project>, and <Worker>. In general, there is no limit on the levels of nesting of elements.

It is possible to characterize three main types of XML documents:

- **Data-centric XML documents.** These documents have many small data items that follow a specific structure and hence may be extracted from a structured database. They are formatted as XML documents in order to exchange them over the Web. These usually follow a *predefined schema* that defines the tag names.
- **Document-centric XML documents.** These are documents with large amounts of text, such as news articles or books. There are few or no structured data elements in these documents.
- **Hybrid XML documents.** These documents may have parts that contain structured data and other parts that are predominantly textual or unstructured. They may or may not have a predefined schema.

[4]SGML (Standard Generalized Markup Language) is a more general language for describing documents and provides capabilities for specifying new tags. However, it is more complex than HTML and XML.

[5]The left and right angled bracket characters (< and >) are reserved characters, as are the ampersand (&), apostrophe ('), and single quotation mark ('). To include them within the text of a document, they must be encoded with escapes as <, >, &, ', and ", respectively.

```
<?xml version="1.0" standalone="yes"?>
    <Projects>
        <Project>
            <Name>ProductX</Name>
            <Number>1</Number>
            <Location>Bellaire</Location>
            <Dept_no>5</Dept_no>
            <Worker>
                <Ssn>123456789</Ssn>
                <Last_name>Smith</Last_name>
                <Hours>32.5</Hours>
            </Worker>
            <Worker>
                <Ssn>453453453</Ssn>
                <First_name>Joyce</First_name>
                <Hours>20.0</Hours>
            </Worker>
        </Project>
        <Project>
            <Name>ProductY</Name>
            <Number>2</Number>
            <Location>Sugarland</Location>
            <Dept_no>5</Dept_no>
            <Worker>
                <Ssn>123456789</Ssn>
                <Hours>7.5</Hours>
            </Worker>
            <Worker>
                <Ssn>453453453</Ssn>
                <Hours>20.0</Hours>
            </Worker>
            <Worker>
                <Ssn>333445555</Ssn>
                <Hours>10.0</Hours>
            </Worker>
        </Project>
    ...
    </Projects>
```

Figure 13.3
A complex XML element called <Projects>.

XML documents that do not follow a predefined schema of element names and corresponding tree structure are known as **schemaless XML documents**. It is important to note that data-centric XML documents can be considered either as semistructured data or as structured data as defined in Section 13.1. If an XML document conforms to a predefined XML schema or DTD (see Section 13.3), then the document can be considered as *structured data*. On the other hand, XML allows

documents that do not conform to any schema; these would be considered as *semistructured data* and are *schemaless XML documents*. When the value of the standalone attribute in an XML document is yes, as in the first line in Figure 13.3, the document is standalone and schemaless.

XML attributes are generally used in a manner similar to how they are used in HTML (see Figure 13.2), namely, to describe properties and characteristics of the elements (tags) within which they appear. It is also possible to use XML attributes to hold the values of simple data elements; however, this is generally not recommended. An exception to this rule is in cases that need to **reference** another element in another part of the XML document. To do this, it is common to use attribute values in one element as the references. This resembles the concept of foreign keys in relational databases, and it is a way to get around the strict hierarchical model that the XML tree model implies. We discuss XML attributes further in Section 13.3 when we discuss XML schema and DTD.

13.3 XML Documents, DTD, and XML Schema

13.3.1 Well-Formed and Valid XML Documents and XML DTD

In Figure 13.3, we saw what a simple XML document may look like. An XML document is **well formed** if it follows a few conditions. In particular, it must start with an **XML declaration** to indicate the version of XML being used as well as any other relevant attributes, as shown in the first line in Figure 13.3. It must also follow the syntactic guidelines of the tree data model. This means that there should be a *single root element*, and every element must include a matching pair of start and end tags *within* the start and end tags *of the parent element*. This ensures that the nested elements specify a well-formed tree structure.

A well-formed XML document is syntactically correct. This allows it to be processed by generic processors that traverse the document and create an internal tree representation. A standard model with an associated set of API (application programming interface) functions called **DOM** (Document Object Model) allows programs to manipulate the resulting tree representation corresponding to a well-formed XML document. However, the whole document must be parsed beforehand when using DOM in order to convert the document to that standard DOM internal data structure representation. Another API called **SAX** (Simple API for XML) allows processing of XML documents on the fly by notifying the processing program through callbacks whenever a start or end tag is encountered. This makes it easier to process large documents and allows for processing of so-called **streaming XML documents**, where the processing program can process the tags as they are encountered. This is also known as **event-based processing**. There are also other specialized processors that work with various programming and scripting languages for parsing XML documents.

A well-formed XML document can be schemaless; that is, it can have any tag names for the elements within the document. In this case, there is no predefined

set of elements (tag names) that a program processing the document knows to expect. This gives the document creator the freedom to specify new elements but limits the possibilities for automatically interpreting the meaning or semantics of the elements within the document.

A stronger criterion is for an XML document to be **valid**. In this case, the document must be well formed, and it must follow a particular schema. That is, the element names used in the start and end tag pairs must follow the structure specified in a separate XML **DTD** (**Document Type Definition**) file or **XML schema file**. We first discuss XML DTD here, and then we give an overview of XML schema in Section 13.3.2. Figure 13.4 shows a simple XML DTD file, which specifies the elements (tag names) and their nested structures. Any valid documents conforming to this DTD should follow the specified structure. A special syntax exists for specifying DTD files, as illustrated in Figure 13.4(a). First, a name is given to the **root tag** of the document, which is called Projects in the first line in Figure 13.4. Then the elements and their nested structure are specified.

When specifying elements, the following notation is used:

- A * following the element name means that the element can be repeated zero or more times in the document. This kind of element is known as an *optional multivalued (repeating) element.*
- A + following the element name means that the element can be repeated one or more times in the document. This kind of element is a *required multivalued (repeating) element.*
- A ? following the element name means that the element can be repeated zero or one times. This kind is an *optional single-valued (nonrepeating) element.*
- An element appearing without any of the preceding three symbols must appear exactly once in the document. This kind is a *required single-valued (nonrepeating) element.*
- The **type** of the element is specified via parentheses following the element. If the parentheses include names of other elements, these latter elements are the *children* of the element in the tree structure. If the parentheses include the keyword #PCDATA or one of the other data types available in XML DTD, the element is a leaf node. PCDATA stands for *parsed character data*, which is roughly similar to a string data type.
- The list of attributes that can appear within an element can also be specified via the keyword !ATTLIST. In Figure 13.3, the Project element has an attribute ProjId. If the type of an attribute is ID, then it can be referenced from another attribute whose type is IDREF within another element. Notice that attributes can also be used to hold the values of simple data elements of type #PCDATA.
- Parentheses can be nested when specifying elements.
- A bar symbol ($e_1 \mid e_2$) specifies that either e_1 or e_2 can appear in the document.

We can see that the tree structure in Figure 13.1 and the XML document in Figure 13.3 conform to the XML DTD in Figure 13.4. To require that an XML document be checked for conformance to a DTD, we must specify this in the

```
(a) <!DOCTYPE Projects [
        <!ELEMENT Projects (Project+)>
        <!ELEMENT Project (Name, Number, Location, Dept_no?, Workers)>
            <!ATTLIST Project
                ProjId ID #REQUIRED>
        <!ELEMENT Name (#PCDATA)>
        <!ELEMENT Number (#PCDATA)
        <!ELEMENT Location (#PCDATA)>
        <!ELEMENT Dept_no (#PCDATA)>
        <!ELEMENT Workers (Worker*)>
        <!ELEMENT Worker (Ssn, Last_name?, First_name?, Hours)>
        <!ELEMENT Ssn (#PCDATA)>
        <!ELEMENT Last_name (#PCDATA)>
        <!ELEMENT First_name (#PCDATA)>
        <!ELEMENT Hours (#PCDATA)>
    ] >
```

```
(b) <!DOCTYPE Company [
        <!ELEMENT Company( (Employee|Department|Project)*)>
        <!ELEMENT Department (DName, Location+)>
            <!ATTLIST Department
                DeptId ID #REQUIRED>

        <!ELEMENT Employee (EName, Job, Salary)>
            <!ATTLIST Project
                EmpId ID #REQUIRED
                DeptId IDREF #REQUIRED>
        <!ELEMENT Project (PName, Location)
            <!ATTLIST Project
                ProjId ID #REQUIRED
                Workers IDREFS #IMPLIED>
        <!ELEMENT DName (#PCDATA)>
        <!ELEMENT EName (#PCDATA)>
        <!ELEMENT PName (#PCDATA)>
        <!ELEMENT Job (#PCDATA)
        <!ELEMENT Location (#PCDATA)>
        <!ELEMENT Salary (#PCDATA)>
    ] >
```

Figure 13.4
(a) An XML DTD file called *Projects*.
(b) An XML DTD file called *Company*.

declaration of the document. For example, we could change the first line in Figure 13.3 to the following:

```
<?xml version = "1.0" standalone = "no"?>
<!DOCTYPE Projects SYSTEM "proj.dtd">
```

When the value of the standalone attribute in an XML document is "no", the document needs to be checked against a separate DTD document or XML schema document (see Section 13.2.2). The DTD file shown in Figure 13.4 should be stored in

the same file system as the XML document and should be given the file name proj.dtd. Alternatively, we could include the DTD document text at the beginning of the XML document itself to allow the checking.

Figure 13.4(b) shows another DTD document called Company to illustrate the use of IDREF. A Company document can have any number of Department, Employee, and Project elements, with IDs DeptID, EmpId, and ProjID, respectively. The Employee element has an attribute DeptId of type IDREF, which is a reference to the Department element where the employee works; this is similar to a foreign key. The Project element has an attribute Workers of type IDREFS, which will hold a list of Employee EmpIDs that work on that project; this is similar to a collection or list of foreign keys. The #IMPLIED keyword means that this attribute is optional. It is also possible to provide a default value for any attribute.

Although XML DTD is adequate for specifying tree structures with required, optional, and repeating elements, and with various types of attributes, it has several limitations. First, the data types in DTD are not very general. Second, DTD has its own special syntax and thus requires specialized processors. It would be advantageous to specify XML schema documents using the syntax rules of XML itself so that the same processors used for XML documents could process XML schema descriptions. Third, all DTD elements are always forced to follow the specified ordering of the document, so unordered elements are not permitted. These drawbacks led to the development of XML schema, a more general but also more complex language for specifying the structure and elements of XML documents.

13.3.2 XML Schema

The **XML schema language** is a standard for specifying the structure of XML documents. It uses the same syntax rules as regular XML documents, so that the same processors can be used on both. To distinguish the two types of documents, we will use the term *XML instance document* or *XML document* for a regular XML document that contains both tag names and data values, and *XML schema document* for a document that specifies an XML schema. An XML schema document would contain only tag names, tree structure information, constraints, and other descriptions but no data values. Figure 13.5 shows an XML schema document corresponding to the COMPANY database shown in Figure 5.5. Although it is unlikely that we would want to display the whole database as a single document, there have been proposals to store data in *native XML* format as an alternative to storing the data in relational databases. The schema in Figure 13.5 would serve the purpose of specifying the structure of the COMPANY database if it were stored in a native XML system. We discuss this topic further in Section 13.4.

As with XML DTD, XML schema is based on the tree data model, with elements and attributes as the main structuring concepts. However, it borrows additional concepts from database and object models, such as keys, references, and identifiers. Here we describe the features of XML schema in a step-by-step manner, referring to the sample XML schema document in Figure 13.5 for illustration. We introduce and describe some of the schema concepts in the order in which they are used in Figure 13.5.

Figure 13.5
An XML schema file called *company*.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">Company Schema (Element Approach) - Prepared by Babak
            Hojabri</xsd:documentation>
    </xsd:annotation>
<xsd:element name="company">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="department" type="Department" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="employee" type="Employee" minOccurs="0" maxOccurs="unbounded">
                <xsd:unique name="dependentNameUnique">
                    <xsd:selector xpath="employeeDependent" />
                    <xsd:field xpath="dependentName" />
                </xsd:unique>
            </xsd:element>
            <xsd:element name="project" type="Project" minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:unique name="departmentNameUnique">
        <xsd:selector xpath="department" />
        <xsd:field xpath="departmentName" />
    </xsd:unique>
    <xsd:unique name="projectNameUnique">
        <xsd:selector xpath="project" />
        <xsd:field xpath="projectName" />
    </xsd:unique>
    <xsd:key name="projectNumberKey">
        <xsd:selector xpath="project" />
        <xsd:field xpath="projectNumber" />
    </xsd:key>
    <xsd:key name="departmentNumberKey">
        <xsd:selector xpath="department" />
        <xsd:field xpath="departmentNumber" />
    </xsd:key>
    <xsd:key name="employeeSSNKey">
        <xsd:selector xpath="employee" />
        <xsd:field xpath="employeeSSN" />
    </xsd:key>
    <xsd:keyref name="departmentManagerSSNKeyRef" refer="employeeSSNKey">
        <xsd:selector xpath="department" />
        <xsd:field xpath="departmentManagerSSN" />
    </xsd:keyref>
```

(continues)

Figure 13.5 (continued)
An XML schema file called *company*.

```
    <xsd:keyref name="employeeDepartmentNumberKeyRef"
        refer="departmentNumberKey">
        <xsd:selector xpath="employee" />
        <xsd:field xpath="employeeDepartmentNumber" />
    </xsd:keyref>
    <xsd:keyref name="employeeSupervisorSSNKeyRef" refer="employeeSSNKey">
        <xsd:selector xpath="employee" />
        <xsd:field xpath="employeeSupervisorSSN" />
    </xsd:keyref>
    <xsd:keyref name="projectDepartmentNumberKeyRef" refer="departmentNumberKey">
        <xsd:selector xpath="project" />
        <xsd:field xpath="projectDepartmentNumber" />
    </xsd:keyref>
    <xsd:keyref name="projectWorkerSSNKeyRef" refer="employeeSSNKey">
        <xsd:selector xpath="project/projectWorker" />
        <xsd:field xpath="SSN" />
    </xsd:keyref>
    <xsd:keyref name="employeeWorksOnProjectNumberKeyRef"
        refer="projectNumberKey">
        <xsd:selector xpath="employee/employeeWorksOn" />
        <xsd:field xpath="projectNumber" />
    </xsd:keyref>
</xsd:element>
<xsd:complexType name="Department">
    <xsd:sequence>
        <xsd:element name="departmentName" type="xsd:string" />
        <xsd:element name="departmentNumber" type="xsd:string" />
        <xsd:element name="departmentManagerSSN" type="xsd:string" />
        <xsd:element name="departmentManagerStartDate" type="xsd:date" />
        <xsd:element name="departmentLocation" type="xsd:string" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Employee">
    <xsd:sequence>
        <xsd:element name="employeeName" type="Name" />
        <xsd:element name="employeeSSN" type="xsd:string" />
        <xsd:element name="employeeSex" type="xsd:string" />
        <xsd:element name="employeeSalary" type="xsd:unsignedInt" />
        <xsd:element name="employeeBirthDate" type="xsd:date" />
        <xsd:element name="employeeDepartmentNumber" type="xsd:string" />
        <xsd:element name="employeeSupervisorSSN" type="xsd:string" />
        <xsd:element name="employeeAddress" type="Address" />
        <xsd:element name="employeeWorksOn" type="WorksOn" minOccurs="1" maxOccurs="unbounded" />
        <xsd:element name="employeeDependent" type="Dependent" minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
</xsd:complexType>
```

Figure 13.5 (continued)
An XML schema file called *company*.

```
<xsd:complexType name="Project">
    <xsd:sequence>
        <xsd:element name="projectName" type="xsd:string" />
        <xsd:element name="projectNumber" type="xsd:string" />
        <xsd:element name="projectLocation" type="xsd:string" />
        <xsd:element name="projectDepartmentNumber" type="xsd:string" />
        <xsd:element name="projectWorker" type="Worker" minOccurs="1" maxOccurs="unbounded" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Dependent">
    <xsd:sequence>
        <xsd:element name="dependentName" type="xsd:string" />
        <xsd:element name="dependentSex" type="xsd:string" />
        <xsd:element name="dependentBirthDate" type="xsd:date" />
        <xsd:element name="dependentRelationship" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Address">
    <xsd:sequence>
        <xsd:element name="number" type="xsd:string" />
        <xsd:element name="street" type="xsd:string" />
        <xsd:element name="city" type="xsd:string" />
        <xsd:element name="state" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Name">
    <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string" />
        <xsd:element name="middleName" type="xsd:string" />
        <xsd:element name="lastName" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Worker">
    <xsd:sequence>
        <xsd:element name="SSN" type="xsd:string" />
        <xsd:element name="hours" type="xsd:float" />
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="WorksOn">
    <xsd:sequence>
        <xsd:element name="projectNumber" type="xsd:string" />
        <xsd:element name="hours" type="xsd:float" />
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

1. **Schema descriptions and XML namespaces.** It is necessary to identify the specific set of XML schema language elements (tags) being used by specifying a file stored at a Web site location. The second line in Figure 13.5 specifies the file used in this example, which is http://www.w3.org/2001/XMLSchema. This is a commonly used standard for XML schema commands. Each such definition is called an **XML namespace** because it defines the set of commands (names) that can be used. The file name is assigned to the variable xsd (XML schema description) using the attribute xmlns (XML namespace), and this variable is used as a prefix to all XML schema commands (tag names). For example, in Figure 13.5, when we write xsd:element or xsd:sequence, we are referring to the definitions of the element and sequence tags as defined in the file http://www.w3.org/2001/XMLSchema.
2. **Annotations, documentation, and language used.** The next couple of lines in Figure 13.5 illustrate the XML schema elements (tags) xsd:annotation and xsd:documentation, which are used for providing comments and other descriptions in the XML document. The attribute xml:lang of the xsd:documentation element specifies the language being used, where en stands for the English language.
3. **Elements and types.** Next, we specify the *root element* of our XML schema. In XML schema, the name attribute of the xsd:element tag specifies the element name, which is called company for the root element in our example (see Figure 13.5). The structure of the company root element can then be specified, which in our example is xsd:complexType. This is further specified to be a sequence of departments, employees, and projects using the xsd:sequence structure of XML schema. It is important to note here that this is not the only way to specify an XML schema for the COMPANY database. We will discuss other options in Section 13.6.
4. **First-level elements in the COMPANY database.** Next, we specify the three first-level elements under the company root element in Figure 13.5. These elements are named employee, department, and project, and each is specified in an xsd:element tag. Notice that if a tag has only attributes and no further subelements or data within it, it can be ended with the backslash symbol (/>) directly instead of having a separate matching end tag. These are called **empty elements**; examples are the xsd:element elements named department and project in Figure 13.5.
5. **Specifying element type and minimum and maximum occurrences.** In XML schema, the attributes type, minOccurs, and maxOccurs in the xsd:element tag specify the type and multiplicity of each element in any document that conforms to the schema specifications. If we specify a type attribute in an xsd:element, the structure of the element must be described separately, typically using the xsd:complexType element of XML schema. This is illustrated by the employee, department, and project elements in Figure 13.5. On the other hand, if no type attribute is specified, the element structure can be defined directly following the tag, as illustrated by the company root element in Figure 13.5. The minOccurs and maxOccurs tags are used for specifying lower

and upper bounds on the number of occurrences of an element in any XML document that conforms to the schema specifications. If they are not specified, the default is exactly one occurrence. These serve a similar role to the *, +, and ? symbols of XML DTD.

6. **Specifying keys.** In XML schema, it is possible to specify constraints that correspond to unique and primary key constraints in a relational database (see Section 5.2.2), as well as foreign keys (or referential integrity) constraints (see Section 5.2.4). The xsd:unique tag specifies elements that correspond to unique attributes in a relational database. We can give each such uniqueness constraint a name, and we must specify xsd:selector and xsd:field tags for it to identify the element type that contains the unique element and the element name within it that is unique via the xpath attribute. This is illustrated by the departmentNameUnique and projectNameUnique elements in Figure 13.5. For specifying **primary keys**, the tag xsd:key is used instead of xsd:unique, as illustrated by the projectNumberKey, departmentNumberKey, and employeeSSNKey elements in Figure 13.5. For specifying **foreign keys**, the tag xsd:keyref is used, as illustrated by the six xsd:keyref elements in Figure 13.5. When specifying a foreign key, the attribute refer of the xsd:keyref tag specifies the referenced primary key, whereas the tags xsd:selector and xsd:field specify the referencing element type and foreign key (see Figure 13.5).

7. **Specifying the structures of complex elements via complex types.** The next part of our example specifies the structures of the complex elements Department, Employee, Project, and Dependent, using the tag xsd:complexType (see Figure 13.5). We specify each of these as a sequence of subelements corresponding to the database attributes of each entity type (see Figure 7.7)by using the xsd:sequence and xsd:element tags of XML schema. Each element is given a name and type via the attributes name and type of xsd:element. We can also specify minOccurs and maxOccurs attributes if we need to change the default of exactly one occurrence. For (optional) database attributes where null is allowed, we need to specify minOccurs = 0, whereas for multivalued database attributes we need to specify maxOccurs = "unbounded" on the corresponding element. Notice that if we were not going to specify any key constraints, we could have embedded the subelements within the parent element definitions directly without having to specify complex types. However, when unique, primary key and foreign key constraints need to be specified; we must define complex types to specify the element structures.

8. **Composite (compound) attributes.** Composite attributes from Figure 9.2 are also specified as complex types in Figure 13.7, as illustrated by the Address, Name, Worker, and WorksOn complex types. These could have been directly embedded within their parent elements.

This example illustrates some of the main features of XML schema. There are other features, but they are beyond the scope of our presentation. In the next section, we discuss the different approaches to creating XML documents from relational databases and storing XML documents.

13.4 Storing and Extracting XML Documents from Databases

Several approaches to organizing the contents of XML documents to facilitate their subsequent querying and retrieval have been proposed. The following are the most common approaches:

1. **Using a file system or a DBMS to store the documents as text.** An XML document can be stored as a text file within a traditional file system. Alternatively, a relational DBMS can be used to store whole XML documents as text fields within the DBMS recordss. This approach can be used if the DBMS has a special module for document processing, and it would work for storing schemaless and document-centric XML documents.
2. **Using a DBMS to store the document contents as data elements.** This approach would work for storing a collection of documents that follow a specific XML DTD or XML schema. Because all the documents have the same structure, one can design a relational database to store the leaf-level data elements within the XML documents. This approach would require mapping algorithms to design a database schema that is compatible with the XML document structure as specified in the XML schema or DTD and to re-create the XML documents from the stored data. These algorithms can be implemented either as an internal DBMS module or as separate middleware that is not part of the DBMS. If all elements in an XML document have IDs, a simple representation would be to have a table with attributes XDOC(CId, PId, Etag, Val) where CID and PId are the parent and child element IDs, Etag is the name of the element of the Cid, and Val is the value if it is a leaf node, assuming all values are the same type.
3. **Designing a specialized system for storing native XML data.** A new type of database system based on the hierarchical (tree) model could be designed and implemented. Such systems are referred to as **native XML DBMS**s. The system would include specialized indexing and querying techniques and would work for all types of XML documents. It could also include data compression techniques to reduce the size of the documents for storage. Tamino by Software AG and the Dynamic Application Platform of eXcelon are two popular products that offer native XML DBMS capability. Oracle also offers a native XML storage option.
4. **Creating or publishing customized XML documents from preexisting relational databases.** Because there are enormous amounts of data already stored in relational databases, parts of this data may need to be formatted as documents for exchanging or displaying over the Web. This approach would use a separate middleware software layer to handle the conversions needed between the relational data and the extracted XML documents. Section 13.6 discusses this approach, in which data-centric XML documents are extracted from existing databases, in more detail. In particular, we show how tree structured documents can be created from flat relational databases that have

been designed using the ER graph-structured data model. Section 13.6.2 discusses the problem of cycles and how to deal with it.

All of these approaches have received considerable attention. We focus on the fourth approach in Section 13.6, because it gives a good conceptual understanding of the differences between the XML tree data model and the traditional database models based on flat files (relational model) and graph representations (ER model). But first we give an overview of XML query languages in Section 13.5.

13.5 XML Languages

There have been several proposals for XML query languages, and two query language standards have emerged. The first is **XPath**, which provides language constructs for specifying path expressions to identify certain nodes (elements) or attributes within an XML document that match specific patterns. The second is **XQuery**, which is a more general query language. XQuery uses XPath expressions but has additional constructs. We give an overview of each of these languages in this section. Then we discuss some additional languages related to HTML in Section 13.5.3.

13.5.1 XPath: Specifying Path Expressions in XML

An XPath expression generally returns a sequence of items that satisfy a certain pattern as specified by the expression. These items are either values (from leaf nodes) or elements or attributes. The most common type of XPath expression returns a collection of element or attribute nodes that satisfy certain patterns specified in the expression. The names in the XPath expression are node names in the XML document tree that are either tag (element) names or attribute names, possibly with additional **qualifier conditions** to further restrict the nodes that satisfy the pattern. Two main **separators** are used when specifying a path: single slash (/) and double slash (//). A single slash before a tag specifies that the tag must appear as a direct child of the previous (parent) tag, whereas a double slash specifies that the tag can appear as a descendant of the previous tag *at any level*. To refer to an attribute name instead of an element (tag) name, the prefix @ is used before the attribute name. Let us look at some examples of XPath as shown in Figure 13.6.

The first XPath expression in Figure 13.6 returns the company root node and all its descendant nodes, which means that it returns the whole XML document. We should note that it is customary to include the file name in the XPath query. This allows us to specify any local file name or even any path name that specifies a file on the Web. For example, if the COMPANY XML document is stored at the location

www.company.com/info.XML

then the first XPath expression in Figure 13.6 can be written as

doc(www.company.com/info.XML)/company

This prefix would also be included in the other examples of XPath expressions.

1. /company
2. /company/department
3. //employee [employeeSalary gt 70000]/employeeName
4. /company/employee [employeeSalary gt 70000]/employeeName
5. /company/project/projectWorker [hours ge 20.0]

Figure 13.6
Some examples of XPath expressions on XML documents that follow the XML schema file *company* in Figure 13.5.

The second example in Figure 13.6 returns all department nodes (elements) and their descendant subtrees. Note that the nodes (elements) in an XML document are ordered, so the XPath result that returns multiple nodes will do so in the same order in which the nodes are ordered in the document tree.

The third XPath expression in Figure 13.6 illustrates the use of //, which is convenient to use if we do not know the full path name we are searching for, but we do know the name of some tags of interest within the XML document. This is particularly useful for schemaless XML documents or for documents with many nested levels of nodes.[6]

The expression returns all employeeName nodes that are direct children of an employee node, such that the employee node has another child element employeeSalary whose value is greater than 70000. This illustrates the use of qualifier conditions, which restrict the nodes selected by the XPath expression to those that satisfy the condition. XPath has a number of comparison operations for use in qualifier conditions, including standard arithmetic, string, and set comparison operations.

The fourth XPath expression in Figure 13.6 should return the same result as the previous one, except that we specified the full path name in this example. The fifth expression in Figure 13.6 returns all projectWorker nodes and their descendant nodes that are children under a path /company/project and have a child node, hours, with a value greater than 20.0 hours.

When we need to include attributes in an XPath expression, the attribute name is prefixed by the @ symbol to distinguish it from element (tag) names. It is also possible to use the **wildcard** symbol *, which stands for any element, as in the following example, which retrieves all elements that are child elements of the root, regardless of their element type. When wildcards are used, the result can be a sequence of different types of elements.

```
/company/*
```

The examples above illustrate simple XPath expressions, where we can only move down in the tree structure from a given node. A more general model for path expressions has been proposed. In this model, it is possible to move in multiple directions from the current node in the path expression. These are known as the

[6]We use the terms *node*, *tag*, and *element* interchangeably here.

axes of an XPath expression. Our examples above used only *three of these axes*: child of the current node (/), descendent or self at any level of the current node (//), and attribute of the current node (@). Other axes include parent, ancestor (at any level), previous sibling (a node at same level to the left), and next sibling (a node at the same level to the right). These axes allow for more complex path expressions.

The main restriction of XPath path expressions is that the path that specifies the pattern also specifies the items to be retrieved. Hence, it is difficult to specify certain conditions on the pattern while separately specifying which result items should be retrieved. The XQuery language separates these two concerns and provides more powerful constructs for specifying queries.

13.5.2 XQuery: Specifying Queries in XML

XPath allows us to write expressions that select items from a tree-structured XML document. XQuery permits the specification of more general queries on one or more XML documents. The typical form of a query in XQuery is known as a **FLWOR expression**, which stands for the five main clauses of XQuery and has the following form:

```
FOR <variable bindings to individual nodes (elements)>
LET <variable bindings to collections of nodes (elements)>
WHERE <qualifier conditions>
ORDER BY <ordering specifications>
RETURN <query result specification>
```

There can be zero or more instances of the FOR clause, as well as of the LET clause in a single XQuery. The WHERE and ORDER BY clauses are optional but can appear at most once, and the RETURN clause must appear exactly once. Let us illustrate these clauses with the following simple example of an XQuery.

```
LET $d : = doc(www.company.com/info.xml)
FOR $x IN $d/company/project[projectNumber = 5]/projectWorker,
        $y IN $d/company/employee
WHERE $x/hours gt 20.0 AND $y.ssn = $x.ssn
ORDER BY $x/hours
RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName,
                    $x/hours </res>
```

1. Variables are prefixed with the $ sign. In the above example, $d, $x, and $y are variables. The LET clause assigns a variable to a particular expression for the rest of the query. In this example, $d is assigned to the document file name. It is possible to have a query that refers to multiple documents by assigning multiple variables in this way.
2. The FOR clause assigns a variable to range over each of the individual elements in a sequence. In our example, the sequences are specified by path expressions. The $x variable ranges over elements that satisfy the path expression $d/company/project[projectNumber = 5]/projectWorker. The $y variable