designers can treat external data sources, such as files or data in nonrelational databases, as if they were "foreign" tables.

- Part 10: SQL/OLB (Object Language Bindings) defines standards for embedding SQL in Java.
- Part 11: SQL/Schemata (Information and Definition Schema) defines a standard catalog interface.
- Part 13: SQL/JRT (Java Routines and Types) defines standards for accessing routines and types in Java.
- Part 14: SQL/XML defines XML-Related Specifications.

The missing numbers cover features such as temporal data, distributed transaction processing, and multimedia data, for which there is as yet no agreement on the standards.

The latest versions of the SQL standard are SQL:2006, which added several features related to XML, and SQL:2008, which introduces a number of extensions to the SQL language.

24.4.2 Database Connectivity Standards

The ODBC standard is a widely used standard for communication between client applications and database systems. ODBC is based on the SQL Call Level Interface (CLI) standards developed by the X/Open industry consortium and the SQL Access Group, but it has several extensions. The ODBC API defines a CLI, an SQL syntax definition, and rules about permissible sequences of CLI calls. The standard also defines conformance levels for the CLI and the SQL syntax. For example, the core level of the CLI has commands to connect to a database, to prepare and execute SQL statements, to get back results or status values, and to manage transactions. The next level of conformance (level 1) requires support for catalog information retrieval and some other features over and above the core-level CLI; level 2 requires further features, such as ability to send and retrieve arrays of parameter values and to retrieve more detailed catalog information.

ODBC allows a client to connect simultaneously to multiple data sources and to switch among them, but transactions on each are independent; ODBC does not support two-phase commit.

A distributed system provides a more general environment than a client–server system. The X/Open consortium has also developed the X/Open XA standards for interoperation of databases. These standards define transaction-management primitives (such as transaction begin, commit, abort, and prepare-to-commit) that compliant databases should provide; a transaction manager can invoke these primitives to implement distributed transactions by two-phase commit. The XA standards are independent of the data model and of the specific interfaces between clients and databases to exchange data. Thus, we can use the XA protocols to implement a distributed transaction system in which a single transac-

Standardization

Standards define the interface of a software system; for example, standards define the syntax and semantics of a programming language, or the functions in an application-program interface, or even a data model (such as the object-oriented database standards). Today, database systems are complex, and are often made up of multiple independently created parts that need to interact. For example, client programs may be created independently of back-end systems, but the two must be able to interact with each other. A company that has multiple heterogeneous database systems may need to exchange data between the databases. Given such a scenario, standards play an important role.

Formal standards are those developed by a standards organization or by industry groups, through a public process. Dominant products sometimes become de facto standards, in that they become generally accepted as standards without any formal process of recognition. Some formal standards, like many aspects of the SQL-92 and SQL:1999 standards, are anticipatory standards that lead the marketplace; they define features that vendors then implement in products. In other cases, the standards, or parts of the standards, are reactionary standards, in that they attempt to standardize features that some vendors have already implemented, and that may even have become de facto standards. SQL-89 was in many ways reactionary, since it standardized features, such as integrity checking, that were already present in the IBM SAA SQL standard and in other databases.

Formal standards committees are typically composed of representatives of the vendors and of members from user groups and standards organizations such as the International Organization for Standardization (ISO) or the American National Standards Institute (ANSI), or professional bodies, such as the Institute of Electrical and Electronics Engineers (IEEE). Formal standards committees meet periodically, and members present proposals for features to be added to or modified in the standard. After a (usually extended) period of discussion, modifications to the proposal, and public review, members vote on whether to accept or reject a feature. Some time after a standard has been defined and implemented, its shortcomings become clear and new requirements become apparent. The process of updating

the standard then begins, and a new version of the standard is usually released after a few years. This cycle usually repeats every few years, until eventually (perhaps many years later) the standard becomes technologically irrelevant, or loses its user base.

The DBTG CODASYL standard for network databases, formulated by the Database Task Group, was one of the early formal standards for databases. IBM database products formerly established de facto standards, since IBM commanded much of the database market. With the growth of relational databases came a number of new entrants in the database business; hence, the need for formal standards arose. In recent years, Microsoft has created a number of specifications that also have become de facto standards. A notable example is ODBC, which is now used in non-Microsoft environments. JDBC, whose specification was created by Sun Microsystems, is another widely used de facto standard.

This section gives a very high-level overview of different standards, concentrating on the goals of the standard. The bibliographical notes at the end of the chapter provide references to detailed descriptions of the standards mentioned in this section.

24.4.1 SQL Standards

Since SQL is the most widely used query language, much work has been done on standardizing it. ANSI and ISO, with the various database vendors, have played a leading role in this work. The SQL-86 standard was the initial version. The IBM Systems Application Architecture (SAA) standard for SQL was released in 1987. As people identified the need for more features, updated versions of the formal SQL standard were developed, called SQL-89 and SQL-92.

The SQL:1999 version of the SQL standard added a variety of features to SQL. We have seen many of these features in earlier chapters. The SQL:2003 version of the SQL standard is a minor extension of the SQL:1999 standard. Some features such as the SQL:1999 OLAP features (Section 5.6.3) were specified as an amendment to the earlier version of the SQL:1999 standard, instead of waiting for the release of SQL:2003.

The SQL:2003 standard was broken into several parts:

- Part 1: SQL/Framework provides an overview of the standard.
- Part 2: SQL/Foundation defines the basics of the standard: types, schemas, tables, views, query and update statements, expressions, security model, predicates, assignment rules, transaction management, and so on.
- Part 3: SQL/CLI (Call Level Interface) defines application program interfaces to SQL.
- Part 4: SQL/PSM (Persistent Stored Modules) defines extensions to SQL to make it procedural.
- Part 9: SQL/MED (Management of External Data) defines standards or interfacing an SQL system to external sources. By writing wrappers, system

tion can access relational as well as object-oriented databases, yet the transaction manager ensures global consistency via two-phase commit.

There are many data sources that are not relational databases, and in fact may not be databases at all. Examples are flat files and email stores. Microsoft's OLE-DB is a C++ API with goals similar to ODBC, but for nondatabase data sources that may provide only limited querying and update facilities. Just like ODBC, OLE-DB provides constructs for connecting to a data source, starting a session, executing commands, and getting back results in the form of a rowset, which is a set of result rows.

However, OLE-DB differs from ODBC in several ways. To support data sources with limited feature support, features in OLE-DB are divided into a number of interfaces, and a data source may implement only a subset of the interfaces. An OLE-DB program can negotiate with a data source to find what interfaces are supported. In ODBC commands are always in SQL. In OLE-DB, commands may be in any language supported by the data source; while some sources may support SQL, or a limited subset of SQL, other sources may provide only simple capabilities such as accessing data in a flat file, without any query capability. Another major difference of OLE-DB from ODBC is that a rowset is an object that can be shared by multiple applications through shared memory. A rowset object can be updated by one application, and other applications sharing that object will get notified about the change.

The Active Data Objects (ADO) API, also created by Microsoft, provides an easy-to-use interface to the OLE-DB functionality, which can be called from scripting languages, such as VBScript and JScript. The newer ADO.NET API is designed for applications written in the .NET languages such as C# and Visual Basic.NET. In addition to providing simplified interfaces, it provides an abstraction called the *DataSet* that permits disconnected data access.

24.4.3 Object Database Standards

Standards in the area of object-oriented databases have so far been driven primarily by OODB vendors. The Object Database Management Group (ODMG) was a group formed by OODB vendors to standardize the data model and language interfaces to OODBs. The C++ language interface specified by ODMG was briefly outlined in Chapter 22. ODMG is no longer active. JDO is a standard for adding persistence to Java.

The Object Management Group (OMG) is a consortium of companies, formed with the objective of developing a standard architecture for distributed software applications based on the object-oriented model. OMG brought out the *Object Management Architecture* (OMA) reference model. The *Object Request Broker* (ORB) is a component of the OMA architecture that provides message dispatch to distributed objects transparently, so the physical location of the object is not important. The Common Object Request Broker Architecture (CORBA) provides a detailed specification of the ORB, and includes an Interface Description Language (IDL), which is used to define the data types used for data interchange. The IDL helps to sup-

port data conversion when data are shipped between systems with different data representations.

Microsoft introduced the Entity data model, which incorporates ideas from the entity-relationship and object-oriented data models, and an approach to integrating querying with the programming language, called Language Integrated Querying or LINQ. These are likely to become de facto standards.

24.4.4 XML-Based Standards

A wide variety of standards based on XML (see Chapter 23) have been defined for a wide variety of applications. Many of these standards are related to e-commerce. They include standards promulgated by nonprofit consortia and corporate-backed efforts to create de facto standards.

RosettaNet, which falls into the former category, is an industry consortium that uses XML-based standards to facilitate supply-chain management in the computer and information technology industries. Supply-chain management refers to the purchases of material and services that an organization needs to function. In contrast, customer-relationship management refers to the front end of a company's interaction, dealing with customers. Supply-chain management requires standardization of a variety of things such as:

- Global company identifier: RosettaNet specifies a system for uniquely identifying companies, using a 9-digit identifier called *Data Universal Numbering System* (DUNS).
- Global product identifier: RosettaNet specifies a 14-digit *Global Trade Item Number* (GTIN) for identifying products and services.
- Global class identifier: This is a 10-digit hierarchical code for classifying products and services called the *United Nations/Standard Product and Services Code* (UN/SPSC).
- Interfaces between trading partners: RosettaNet *Partner Interface Processes* (PIPs) define business processes between partners. PIPs are system-to-system XML-based dialogs: They define the formats and semantics of business documents involved in a process and the steps involved in completing a transaction. Examples of steps could include getting product and service information, purchase orders, order invoicing, payment, order status requests, inventory management, post-sales support including service warranty, and so on. Exchange of design, configuration, process, and quality information is also possible to coordinate manufacturing activities across organizations.

Participants in electronic marketplaces may store data in a variety of database systems. These systems may use different data models, data formats, and data types. Furthermore, there may be semantic differences (metric versus English measure, distinct monetary currencies, and so forth) in the data. Standards for electronic marketplaces include methods for *wrapping* each of these heteroge-

neous systems with an XML schema. These XML *wrappers* form the basis of a unified view of data across all of the participants in the marketplace.

Simple Object Access Protocol (SOAP) is a remote procedure call standard that uses XML to encode data (both parameters and results), and uses HTTP as the transport protocol; that is, a procedure call becomes an HTTP request. SOAP is backed by the World Wide Web Consortium (W3C) and has gained wide acceptance in industry. SOAP can be used in a variety of applications. For instance, in business-to-business e-commerce, applications running at one site can access data from and execute actions at other sites through SOAP.

SOAP and Web services were described in more detail in Section 23.7.3.

23.7.3 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.

23.4.2 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 (see Section 22.6). 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 23.11, 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 23.6, 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 instuctors attribute of course elements.

- The | operator allows expression results to be unioned. For example, given data using the schema from Figure 23.11, 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 is 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.

23.4.3 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.

23.4.3.1 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 3 credits, shown below, is based on the XML document of Figure 23.11, 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 above 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.