22 INTERNET DATABASES

He profits most who serves best.

—Motto for Rotary International

The proliferation of computer networks, including the Internet and corporate 'intranets,' has enabled users to access a large number of data sources. This increased access to databases is likely to have a great practical impact; data and services can now be offered directly to customers in ways that were impossible until recently. **Electronic commerce** applications cover a broad spectrum; examples include purchasing books through a Web retailer such as Amazon.com, engaging in online auctions at a site such as eBay, and exchanging bids and specifications for products between companies. The emergence of standards such as XML for describing content (in addition to the presentation aspects) of documents is likely to further accelerate the use of the Web for electronic commerce applications.

While the first generation of Internet sites were collections of HTML files—HTML is a standard for describing how a file should be displayed—most major sites today store a large part (if not all) of their data in database systems. They rely upon DBMSs to provide fast, reliable responses to user requests received over the Internet; this is especially true of sites for electronic commerce. This unprecedented access will lead to increased and novel demands upon DBMS technology. The impact of the Web on DBMSs, however, goes beyond just a new source of large numbers of concurrent queries: The presence of large collections of unstructured text documents and partially structured HTML and XML documents and new kinds of queries such as keyword search challenge DBMSs to significantly expand the data management features they support. In this chapter, we discuss the role of DBMSs in the Internet environment and the new challenges that arise.

We introduce the World Wide Web, Web browsers, Web servers, and the HTML markup language in Section 22.1. In Section 22.2, we discuss alternative architectures for making databases accessible through the Web. We discuss XML, an emerging standard for document description that is likely to supersede HTML, in Section 22.3. Given the proliferation of text documents on the Web, searching them for user-specified keywords is an important new query type. *Boolean* keyword searches ask for documents containing a specified boolean combination of keywords. *Ranked* keyword searches ask for documents that are most relevant to a given list of keywords. We

consider indexing techniques to support boolean keyword searches in Section 22.4 and techniques to support ranked keyword searches in Section 22.5.

22.1 THE WORLD WIDE WEB

The Web makes it possible to access a file anywhere on the Internet. A file is identified by a **universal resource locator (URL)**:

`http://www.informatik.uni-trier.de/~ley/db/index.html`

This URL identifies a file called `index.html`, stored in the directory `~ley/db/` on machine `www.informatik.uni-trier.de`. This file is a document formatted using **HyperText Markup Language (HTML)** and contains several **links** to other files (identified through their URLs).

The formatting commands are interpreted by a **Web browser** such as Microsoft's Internet Explorer or Netscape Navigator to display the document in an attractive manner, and the user can then navigate to other related documents by choosing links. A collection of such documents is called a **Web site** and is managed using a program called a **Web server**, which accepts URLs and returns the corresponding documents. Many organizations today maintain a Web site. (Incidentally, the URL shown above is the entry point to Michael Ley's Databases and Logic Programming (DBLP) Web site, which contains information on database and logic programming research publications. It is an invaluable resource for students and researchers in these areas.) The **World Wide Web**, or **Web**, is the collection of Web sites that are accessible over the Internet.

An HTML link contains a URL, which identifies the site containing the linked file. When a user clicks on a link, the Web browser connects to the Web server at the destination Web site using a connection protocol called **HTTP** and submits the link's URL. When the browser receives a file from a Web server, it checks the file type by examining the extension of the file name. It displays the file according to the file's type and if necessary calls an application program to handle the file. For example, a file ending in `.txt` denotes an unformatted text file, which the Web browser displays by interpreting the individual ASCII characters in the file. More sophisticated document structures can be encoded in HTML, which has become a standard way of structuring Web pages for display. As another example, a file ending in `.doc` denotes a Microsoft Word document and the Web browser displays the file by invoking Microsoft Word.

22.1.1 Introduction to HTML

HTML is a simple language used to describe a document. It is also called a **markup language** because HTML works by augmenting regular text with 'marks' that hold special meaning for a Web browser handling the document. Commands in the language

```
<HTML>
<HEAD></HEAD>
<BODY>
Science:
<UL>
  <LI>Author: Richard Feynman</LI>
  <LI>Title:  The Character of Physical Law</LI>
  <LI>Published 1980</LI>
  <LI>Hardcover</LI>
</UL>
Fiction:
<UL>
  <LI>Author: R.K. Narayan</LI>
  <LI>Title:  Waiting for the Mahatma</LI>
  <LI>Published 1981</LI>
<UL>
  <LI>Name:  R.K. Narayan</LI>
  <LI>Title: The English Teacher</LI>
  <LI>Published 1980</LI>
  <LI>Paperback</LI>
</UL>
</BODY>
</HTML>
```

Figure 22.1 Book Listing in HTML

are called **tags** and they consist (usually) of a **start tag** and an **end tag** of the form `<TAG>` and `</TAG>`, respectively. For example, consider the HTML fragment shown in Figure 22.1. It describes a Web page that shows a list of books. The document is enclosed by the tags `<HTML>` and `</HTML>`, marking it as an HTML document. The remainder of the document—enclosed in `<BODY>` ... `</BODY>`—contains information about three books. Data about each book is represented as an unordered list (`UL`) whose entries are marked with the `LI` tag. HTML defines the set of valid tags as well as the meaning of the tags. For example, HTML specifies that the tag `<TITLE>` is a valid tag that denotes the title of the document. As another example, the tag `<UL>` always denotes an unordered list.

Audio, video, and even programs (written in Java, a highly portable language) can be included in HTML documents. When a user retrieves such a document using a suitable browser, images in the document are displayed, audio and video clips are played, and embedded programs are executed at the user's machine; the result is a rich multimedia presentation. The ease with which HTML documents can be created—

there are now visual editors that automatically generate HTML—and accessed using Internet browsers has fueled the explosive growth of the Web.

22.1.2 Databases and the Web

The Web is the cornerstone of electronic commerce. Many organizations offer products through their Web sites, and customers can place orders by visiting a Web site. For such applications a URL must identify more than just a file, however rich the contents of the file; a URL must provide an entry point to services available on the Web site. It is common for a URL to include a form that users can fill in to describe what they want. If the requested URL identifies a form, the Web server returns the form to the browser, which displays the form to the user. After the user fills in the form, the form is returned to the Web server, and the information filled in by the user can be used as parameters to a program executing at the same site as the Web server.

The use of a Web browser to invoke a program at a remote site leads us to the role of databases on the Web: The invoked program can generate a request to a database system. This capability allows us to easily place a database on a computer network, and make services that rely upon database access available over the Web. This leads to a new and rapidly growing source of concurrent requests to a DBMS, and with thousands of concurrent users routinely accessing popular Web sites, new levels of scalability and robustness are required.

The diversity of information on the Web, its distributed nature, and the new uses that it is being put to lead to challenges for DBMSs that go beyond simply improved performance in traditional functionality. For instance, we require support for queries that are run periodically or continuously and that access data from several distributed sources. As an example, a user may want to be notified whenever a new item meeting some criteria (e.g., a Peace Bear Beanie Baby toy costing less than $15) is offered for sale at one of several Web sites. Given many such user profiles, how can we efficiently monitor them and notify users promptly as the items they are interested in become available? As another instance of a new class of problems, the emergence of the XML standard for describing data leads to challenges in managing and querying XML data (see Section 22.3).

22.2 ARCHITECTURE

To execute a program at the Web server's site, the server creates a new process and communicates with this process using the **common gateway interface (CGI)** protocol. The results of the program can be used to create an HTML document that is returned to the requestor. Pages that are computed in this manner at the time they

```
<HTML><HEAD><TITLE>The Database Bookstore</TITLE></HEAD>
<BODY>
<FORM action="find_books.cgi"  method=post>
   Type an author name:
   <INPUT type="text" name="authorName" size=30 maxlength=50>
   <INPUT type="submit" value="Send it">
   <INPUT type="reset" value="Clear form">
</FORM>
</BODY></HTML>
```

Figure 22.2 A Sample Web Page with Form Input

are requested are called **dynamic pages**; pages that exist and are simply delivered to the Web browser are called **static pages**.

As an example, consider the sample page shown in Figure 22.2. This Web page contains a form where a user can fill in the name of an author. If the user presses the 'Send it' button, the Perl script 'findBooks.cgi' mentioned in Figure 22.2 is executed as a separate process. The CGI protocol defines how the communication between the form and the script is performed. Figure 22.3 illustrates the processes created when using the CGI protocol.

Figure 22.4 shows an example CGI script, written in Perl. We have omitted error-checking code for simplicity. Perl is an interpreted language that is often used for CGI scripting and there are many Perl libraries called **modules** that provide high-level interfaces to the CGI protocol. We use two such libraries in our example: DBI and CGI. DBI is a database-independent API for Perl that allows us to abstract from the DBMS being used—DBI plays the same role for Perl as JDBC does for Java. Here we use DBI as a bridge to an ODBC driver that handles the actual connection to the database. The CGI module is a convenient collection of functions for creating CGI scripts. In part 1 of the sample script, we extract the content of the HTML form as follows:

```
$authorName = $dataIn->param('authorName');
```

Note that the parameter name `authorName` was used in the form in Figure 22.2 to name the first input field. In part 2 we construct the actual SQL command in the variable `$sql`. In part 3 we start to assemble the page that is returned to the Web browser. We want to display the result rows of the query as entries in an unordered list, and we start the list with its start tag `<UL>`. Individual list entries will be enclosed by the `<LI>` tag. Conveniently, the CGI protocol abstracts the actual implementation of how the Web page is returned to the Web browser; the Web page consists simply of the

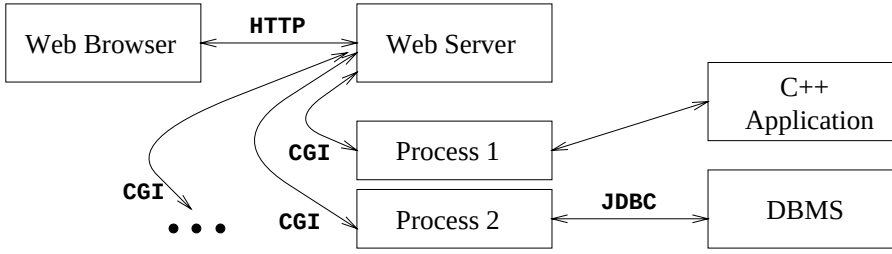

Figure 22.3 Process Structure with CGI Scripts

output of our program. Thus, everything that the script writes in `print`-statements will be part of the dynamically constructed Web page that is returned to the Web browser. Part 4 establishes the database connection and prepares and executes the SQL statement that we stored in the variable `$sql` in part 2. In part 5, we fetch the result of the query, one row at a time, and append each row to the output. Part 6 closes the connection to the database system and we finish in part 7 by appending the closing format tags to the resulting page.

Alternative protocols, in which the program invoked by a request is executed within the Web server process, have been proposed by Microsoft (*Internet Server API (ISAPI)*) and by Netscape (*Netscape Server API (NSAPI)*). Indeed, the TPC-C benchmark has been executed, with good results, by sending requests from 1,500 PC clients to a Web server and through it to an SQL database server.

22.2.1 Application Servers and Server-Side Java

In the previous section, we discussed how the CGI protocol can be used to dynamically assemble Web pages whose content is computed on demand. However, since each page request results in the creation of a new process this solution does not scale well to a large number of simultaneous requests. This performance problem led to the development of specialized programs called **application servers**. An application server has pre-forked threads or processes and thus avoids the startup cost of creating a new process for each request. Application servers have evolved into flexible middle tier packages that provide many different functions in addition to eliminating the process-creation overhead:

- **Integration of heterogeneous data sources:** Most companies have data in many different database systems, from legacy systems to modern object-relational systems. Electronic commerce applications require integrated access to all these data sources.
- **Transactions involving several data sources:** In electronic commerce applications, a user transaction might involve updates at several data sources. An

```
#!/usr/bin/perl
use DBI; use CGI;

### part 1
$dataIn = new CGI;
$dataIn->header();
$authorName = $dataIn->param(‘authorName’);

### part 2
$sql = "SELECT authorName, title FROM books ";
$sql += "WHERE authorName = " + $authorName;

### part 3
print "<HTML><TITLE>Results:</TITLE><UL> Results:";

### part 4
$dbh = DBI->connect( ‘DBI:ODBC:BookStore’, ‘webuser’, ‘password’);
$sth = $dbh->prepare($sql);
$sth->execute;

### part 5
while ( @row = $sth->fetchrow ) {
    print "<LI> @row </LI> \n";
}

### part 6
$sth->finish;
$dbhandle->disconnect;

### part 7
print "</UL></HTML>";
exit;
```

Figure 22.4 A Simple Perl Script

An example of a real application server—IBM WebSphere: IBM WebSphere is an application server that provides a wide range of functionality. It includes a full-fledged Web server and supports dynamic Web page generation. WebSphere includes a Java Servlet run time environment that allows users to extend the functionality of the server. In addition to Java Servlets, Websphere supports other Web technologies such as Java Server Pages and JavaBeans. It includes a connection manager that handles a pool of relational database connections and caches intermediate query results.

application server can ensure transactional semantics across data sources by providing atomicity, isolation, and durability. The **transaction boundary** is the point at which the application server provides transactional semantics. If the transaction boundary is at the application server, very simple client programs are possible.

- **Security:** Since the users of a Web application usually include the general population, database access is performed using a general-purpose user identifier that is known to the application server. While communication between the server and the application at the server side is usually not a security risk, communication between the client (Web browser) and the Web server could be a security hazard. Encryption is usually performed at the Web server, where a secure protocol (in most cases the **Secure Sockets Layer (SSL)** protocol) is used to communicate with the client.
- **Session management:** Often users engage in business processes that take several steps to complete. Users expect the system to maintain continuity during a session, and several session identifiers such as **cookies**, URL extensions, and hidden fields in HTML forms can be used to identify a session. Application servers provide functionality to detect when a session starts and ends and to keep track of the sessions of individual users.

A possible architecture for a Web site with an application server is shown in Figure 22.5. The client (a Web browser) interacts with the Web server through the HTTP protocol. The Web server delivers static HTML or XML pages directly to the client. In order to assemble dynamic pages, the Web server sends a request to the application server. The application server contacts one or more data sources to retrieve necessary data or sends update requests to the data sources. After the interaction with the data sources is completed, the application server assembles the Web page and reports the result to the Web server, which retrieves the page and delivers it to the client.

The execution of business logic at the Web server's site, or **server-side processing**, has become a standard model for implementing more complicated business processes on the Internet. There are many different technologies for server-side processing and

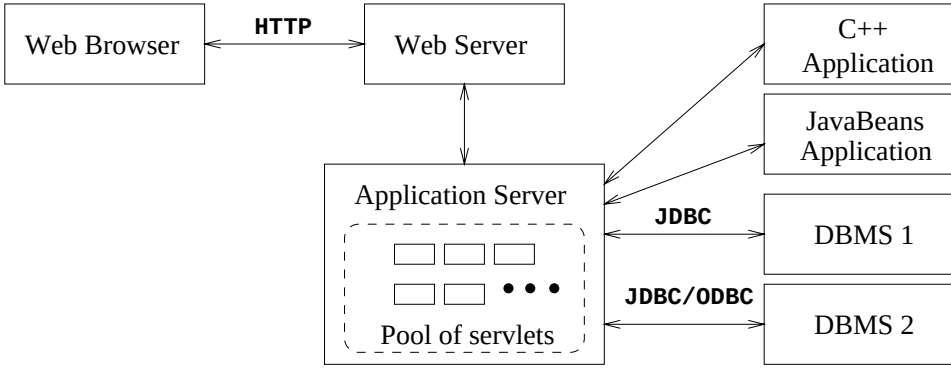

Figure 22.5 Process Structure in the Application Server Architecture

we only mention a few in this section; the interested reader is referred to the references at the end of the chapter.

The **Java Servlet API** allows Web developers to extend the functionality of a Web server by writing small Java programs called **servlets** that interact with the Web server through a well-defined API. A servlet consists of mostly business logic and routines to format relatively small datasets into HTML. Java servlets are executed in their own threads. Servlets can continue to execute even after the client request that led to their invocation is completed and can thus maintain persistent information between requests. The Web server or application server can manage a pool of servlet threads, as illustrated in Figure 22.5, and can therefore avoid the overhead of process creation for each requests. Since servlets are written in Java, they are portable between Web servers and thus allow platform-independent development of server-side applications.

Server-side applications can also be written using JavaBeans. **JavaBeans** are reusable software components written in Java that perform well-defined tasks and can be conveniently packaged and distributed (together with any Java classes, graphics, and other files they need) in **JAR** files. JavaBeans can be assembled to create larger applications and can be easily manipulated using visual tools.

Java Server Pages (JSP) are yet another platform-independent alternative for generating dynamic content on the server side. While servlets are very flexible and powerful, slight modifications, for example in the appearance of the output page, require the developer to change the servlet and to recompile the changes. JSP is designed to separate application logic from the appearance of the Web page, while at the same time simplifying and increasing the speed of the development process. JSP separates content from presentation by using special HTML tags inside a Web page to generate dynamic content. The Web server interprets these tags and replaces them with dynamic content before returning the page to the browser.

For example, consider the following Web page that includes JSP commands:

```
<HTML>
<H1>Hello</H1>
<P>Today is </P>
<jsp:useBean id=="clock" class=="calendar.jspCalendar" />
<UL>
  <LI>Day: <%==clock.getDayOfMonth() %>
  <LI>Year: <%==clock.getYear() %>
</UL>
</HTML>
```

We first load a JavaBean component through the tag `<jsp:useBean>` and then evaluate the `getDayOfMonth` member functions of the bean as marked in the directive `<%==clock.getDayOfMonth() %>`.

The technique of including proprietary markup tags into an HTML file and dynamically evaluating the contents of these tags while assembling the answer page is used in many application servers. Such pages are generally known as **HTML templates**. For example, Cold Fusion is an application server that allows special markup tags that can include database queries. The following code fragment shows an example query:

```
<cfquery name="listBooks" datasource="books">
  select * from books
</cfquery>
```

22.3 BEYOND HTML

While HTML is adequate to represent the structure of documents for display purposes, the features of the language are not sufficient to represent the structure of data within a document for more general applications than a simple display. For example, we can send the HTML document shown in Figure 22.1 to another application and the application can display the information about our books, but using the HTML tags the application cannot distinguish the first names of the authors from their last names. (The application can try to recover such information by looking at the text inside the tags, but that defeats the purpose of structuring the data using tags.) Thus, HTML is inadequate for the exchange of complex documents containing product specifications or bids, for example.

Extensible Markup Language (XML) is a markup language that was developed to remedy the shortcomings of HTML. In contrast to having a fixed set of tags whose meaning is fixed by the language (as in HTML), XML allows the user to define new collections of tags that can then be used to structure any type of data or document the

The design goals of XML: XML was developed starting in 1996 by a working group under guidance of the World Wide Web Consortium (W3C) XML Special Interest Group. The design goals for XML included the following:

1. XML should be compatible with SGML.
2. It should be easy to write programs that process XML documents.
3. The design of XML should be formal and concise.

user wishes to transmit. XML is an important bridge between the document-oriented view of data implicit in HTML and the schema-oriented view of data that is central to a DBMS. It has the potential to make database systems more tightly integrated into Web applications than ever before.

XML emerged from the confluence of two technologies, SGML and HTML. The **Standard Generalized Markup Language (SGML)** is a metalanguage that allows the definition of data and document interchange languages such as HTML. The SGML standard was published in 1988 and many organizations that manage a large number of complex documents have adopted it. Due to its generality, SGML is complex and requires sophisticated programs to harness its full potential. XML was developed to have much of the power of SGML while remaining relatively simple. Nonetheless, XML, like SGML, allows the definition of new document markup languages.

Although XML does not prevent a user from designing tags that encode the display of the data in a Web browser, there is a style language for XML called **Extensible Style Language (XSL)**. XSL is a standard way of describing how an XML document that adheres to a certain vocabulary of tags should be displayed.

22.3.1 Introduction to XML

The short introduction to XML given in this section is not complete, and the references at the end of this chapter provide starting points for the interested reader. We will use the small XML document shown in Figure 22.6 as an example.

- **Elements**. Elements, also called tags, are the primary building blocks of an XML document. The start of the content of an element `ELM` is marked with `<ELM>`, which is called the **start tag**, and the end of the content end is marked with `</ELM>`, called the **end tag**. In our example document, the element `BOOKLIST` encloses all information in the sample document. The element `BOOK` demarcates all data associated with a single book. XML elements are case sensitive: the element `<BOOK>` is different from `<Book>`. Elements must be properly nested. Start tags

that appear inside the content of other tags must have a corresponding end tag. For example, consider the following XML fragment:

```
<BOOK>
  <AUTHOR>
    <FIRSTNAME>Richard</FIRSTNAME><LASTNAME>Feynman</LASTNAME>
  </AUTHOR>
</BOOK>
```

The element `AUTHOR` is completely nested inside the element `BOOK`, and both the elements `LASTNAME` and `FIRSTNAME` are nested inside the element `AUTHOR`.

- **Attributes**. An element can have descriptive attributes that provide additional information about the element. The values of attributes are set inside the start tag of an element. For example, let `ELM` denote an element with the attribute `att`. We can set the value of `att` to `value` through the following expression: `<ELM att="value">`. All attribute values must be enclosed in quotes. In Figure 22.6, the element `BOOK` has two attributes. The attribute `genre` indicates the genre of the book (science or fiction) and the attribute `format` indicates whether the book is a hardcover or a paperback.

- **Entity references**. Entities are shortcuts for portions of common text or the content of external files and we call the usage of an entity in the XML document an **entity reference**. Wherever an entity reference appears in the document, it is textually replaced by its content. Entity references start with a '`&`' and end with a '`;`' . There are five predefined entities in XML that are placeholders for characters with special meaning in XML. For example, the < character that marks the beginning of an XML command is reserved and has to be represented by the entity `lt`. The other four reserved characters are &, >, ", and ', and they are represented by the entities `amp`, `gt`, `quot`, and `apos`. For example, the text '1 < 5' has to be encoded in an XML document as follows: `&apos;1&lt;5&apos;`. We can also use entities to insert arbitrary Unicode characters into the text. **Unicode** is a standard for character representations, and is similar to ASCII. For example, we can display the Japanese Hiragana character 'a' using the entity reference `&#x3042`.

- **Comments.** We can insert comments anywhere in an XML document. Comments start with `<!-` and end with `->`. Comments can contain arbitrary text except the string `--`.

- **Document type declarations (DTDs)**. In XML, we can define our own markup language. A DTD is a set of rules that allows us to specify our own set of elements, attributes, and entities. Thus, a DTD is basically a grammar that indicates what tags are allowed, in what order they can appear, and how they can be nested. We will discuss DTDs in detail in the next section.

We call an XML document **well-formed** if it does not have an associated DTD but follows the following structural guidelines:

```
<?XML version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE BOOKLIST SYSTEM "books.dtd">
<BOOKLIST>
<BOOK genre="Science" format="Hardcover">
  <AUTHOR>
    <FIRSTNAME>Richard</FIRSTNAME><LASTNAME>Feynman</LASTNAME>
  </AUTHOR>
  <TITLE>The Character of Physical Law</TITLE>
  <PUBLISHED>1980</PUBLISHED>
</BOOK>
<BOOK genre="Fiction">
  <AUTHOR>
    <FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
  </AUTHOR>
  <TITLE>Waiting for the Mahatma</TITLE>
  <PUBLISHED>1981</PUBLISHED>
</BOOK>
<BOOK genre="Fiction">
  <AUTHOR>
    <FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
  </AUTHOR>
  <TITLE>The English Teacher</TITLE>
  <PUBLISHED>1980</PUBLISHED>
</BOOK>
</BOOKLIST>
```

Figure 22.6 Book Information in XML

- The document starts with an XML declaration. An example of an XML declaration is the first line of the XML document shown in Figure 22.6.
- There is a root element that contains all the other elements. In our example, the root element is the element `BOOKLIST`.
- All elements must be properly nested. This requirement states that start and end tags of an element must appear within the same enclosing element.

22.3.2 XML DTDs

A DTD is a set of rules that allows us to specify our own set of elements, attributes, and entities. A DTD specifies which elements we can use and constraints on these elements, e.g., how elements can be nested and where elements can appear in the

```
<!DOCTYPE BOOKLIST [
  <!ELEMENT BOOKLIST (BOOK)*>
  <!ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?)>
    <!ELEMENT AUTHOR (FIRSTNAME,LASTNAME)>
      <!ELEMENT FIRSTNAME (#PCDATA)>
      <!ELEMENT LASTNAME (#PCDATA)>
    <!ELEMENT TITLE (#PCDATA)>
    <!ELEMENT PUBLISHED (#PCDATA)>
  <!ATTLIST BOOK genre (Science|Fiction) #REQUIRED>
  <!ATTLIST BOOK format (Paperback|Hardcover) "Paperback">
]>
```

Figure 22.7 Bookstore XML DTD

document. We will call a document **valid** if there is a DTD associated with it and the document is structured according to the rules set by the DTD. In the remainder of this section, we will use the example DTD shown in Figure 22.7 to illustrate how to construct DTDs.

A DTD is enclosed in `<!DOCTYPE name [DTDdeclaration]>`, where `name` is the name of the outermost enclosing tag, and DTDdeclaration is the text of the rules of the DTD. The DTD starts with the outermost element, also called the root element, which is `BOOKLIST` in our example. Consider the next rule:

```
<!ELEMENT BOOKLIST (BOOK)*>
```

This rule tells us that the element `BOOKLIST` consists of zero or more `BOOK` elements. The `*` after `BOOK` indicates how many `BOOK` elements can appear inside the `BOOKLIST` element. A `*` denotes zero or more occurrences, a `+` denotes one or more occurrences, and a `?` denotes zero or one occurrence. For example, if we want to ensure that a `BOOKLIST` has at least one book, we could change the rule as follows:

```
<!ELEMENT BOOKLIST (BOOK)+>
```

Let us look at the next rule:

```
<!ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?)>
```

This rule states that a `BOOK` element contains a `NAME` element, a `TITLE` element, and an optional `PUBLISHED` element. Note the use of the `?` to indicate that the information is optional by having zero or one occurrence of the element. Let us move ahead to the following rule:

```
<!ELEMENT LASTNAME (#PCDATA)>
```

Until now we only considered elements that contained other elements. The above rule states that `LASTNAME` is an element that does not contain other elements, but contains actual text. Elements that only contain other elements are said to have **element content**, whereas elements that also contain `#PCDATA` are said to have **mixed content**. In general, an element type declaration has the following structure:

```
<!ELEMENT (contentType) >
```

The are five possible content types:

- Other elements.
- The special symbol `#PCDATA`, which indicates (parsed) character data.
- The special symbol `EMPTY`, which indicates that the element has no content. Elements that have no content are not required to have an end tag.
- The special symbol `ANY`, which indicates that any content is permitted. This content should be avoided whenever possible since it disables all checking of the document structure inside the element.
- A **regular expression** constructed from the above four choices. A regular expression is one of the following:
  - exp1, exp2, exp3: A list of regular expressions.
  - exp$*$: An optional expression (zero or more occurrences).
  - exp?: An optional expression (zero or one occurrences).
  - exp+: A mandatory expression (one or more occurrences).
  - exp1 | exp2: exp1 or exp2.

Attributes of elements are declared outside of the element. For example, consider the following attribute declaration from Figure 22.7.

```
<!ATTLIST BOOK genre (Science|Fiction) #REQUIRED>
```

This XML DTD fragment specifies the attribute `genre`, which is an attribute of the element `BOOK`. The attribute can take two values: Science or Fiction. Each `BOOK` element must be described in its start tag by a `genre` attribute since the attribute is required as indicated by `#REQUIRED`. Let us look at the general structure of a DTD attribute declaration:

```
<!ATTLIST elementName (attName attType default)+ >
```

The keyword `ATTLIST` indicates the beginning of an attribute declaration. The string `elementName` is the name of the element that the following attribute definition is associated with. What follows is the declaration of one or more attributes. Each attribute has a name as indicated by `attName` and a type as indicated by `attType`. XML defines several possible types for an attribute. We only discuss **string types** and **enumerated types** here. An attribute of type string can take any string as a value. We can declare such an attribute by setting its type field to `CDATA`. For example, we can declare a third attribute of type string of the element `BOOK` as follows:

```
<!ATTLIST BOOK edition CDATA "1">
```

If an attribute has an enumerated type, we list all its possible values in the attribute declaration. In our example, the attribute `genre` is an enumerated attribute type; its possible attribute values are 'Science' and 'Fiction'.

The last part of an attribute declaration is called its **default specification**. The XML DTD in Figure 22.7 shows two different default specifications: `#REQUIRED` and the string 'Paperback'. The default specification `#REQUIRED` indicates that the attribute is required and whenever its associated element appears somewhere in the XML document a value for the attribute must be specified. The default specification indicated by the string 'Paperback' indicates that the attribute is not required; whenever its associated element appears without setting a value for the attribute, the attribute automatically takes the value 'Paperback'. For example, we can make the attribute value 'Science' the default value for the `genre` attribute as follows:

```
<!ATTLIST BOOK genre (Science|Fiction) "Science">
```

The complete XML DTD language is much more powerful than the small part that we have explained. The interested reader is referred to the references at the end of the chapter.

22.3.3 Domain-Specific DTDs

Recently, DTDs have been developed for several specialized domains—including a wide range of commercial, engineering, financial, industrial, and scientific domains—and a lot of the excitement about XML has its origins in the belief that more and more standardized DTDs will be developed. Standardized DTDs would enable seamless data exchange between heterogeneous sources, a problem that is solved today either by implementing specialized protocols such as **Electronic Data Interchange (EDI)** or by implementing ad hoc solutions.

Even in an environment where all XML data is valid, it is not possible to straightforwardly integrate several XML documents by matching elements in their DTDs because

even when two elements have identical names in two different DTDs, the meaning of the elements could be completely different. If both documents use a single, standard DTD we avoid this problem. The development of standardized DTDs is more a social process than a hard research problem since the major players in a given domain or industry segment have to collaborate.

For example, the **mathematical markup language (MathML)** has been developed for encoding mathematical material on the Web. There are two types of MathML elements. The 28 **presentation elements** describe the layout structure of a document; examples are the `mrow` element, which indicates a horizontal row of characters, and the `msup` element, which indicates a base and a subscript. The 75 **content elements** describe mathematical concepts. An example is the `plus` element which denotes the addition operator. (There is a third type of element, the `math` element, that is used to pass parameters to the MathML processor.) MathML allows us to encode mathematical objects in both notations since the requirements of the user of the objects might be different. Content elements encode the precise mathematical meaning of an object without ambiguity and thus the description can be used by applications such as computer algebra systems. On the other hand, good notation can suggest the logical structure to a human and can emphasize key aspects of an object; presentation elements allow us to describe mathematical objects at this level.

For example, consider the following simple equation:

$$x^2 - 4x - 32 = 0$$

Using presentation elements, the equation is represented as follows:

```
<mrow>
  <mrow> <msup><mi>x</mi><mn>2</mn></msup>
         <mo>-</mo>
         <mrow><mn>4</mn><mo>&invisibletimes;</mo><mi>x</mi></mrow>
         <mo>-</mo><mn>32</mn>
  </mrow><mo>=</mo><mn>0</mn>
</mrow>
```

Using content elements, the equation is described as follows:

```
<reln><eq/>
  <apply>
    <minus/>
    <apply> <power/> <ci>x</ci> <cn>2</cn> </apply>
    <apply> <times/> <cn>4</cn> <ci>x</ci> </apply>
    <cn>32</cn>
  </apply> <cn>0</cn>
</reln>
```

Note the additional power that we gain from using MathML instead of encoding the formula in HTML. The common way of displaying mathematical objects inside an HTML object is to include images that display the objects, for example as in the following code fragment:

```
<IMG SRC="images/equation.gif" ALT=" x^2 - 4x - 32 = 10 " >
```

The equation is encoded inside an `IMG` tag with an alternative display format specified in the `ALT` tag. Using this encoding of a mathematical object leads to the following presentation problems. First, the image is usually sized to match a certain font size and on systems with other font sizes the image is either too small or too large. Second, on systems with a different background color the picture does not blend into the background and the resolution of the image is usually inferior when printing the document. Apart from problems with changing presentations, we cannot easily search for a formula or formula fragments on a page, since there is no specific markup tag.

22.3.4 XML-QL: Querying XML Data

Given that data is encoded in a way that reflects (a considerable amount of) structure in XML documents, we have the opportunity to use a high-level language that exploits this structure to conveniently retrieve data from within such documents. Such a language would bring XML data management much closer to database management than the text-oriented paradigm of HTML documents. Such a language would also allow us to easily translate XML data between different DTDs, as is required for integrating data from multiple sources.

At the time of writing of this chapter (summer of 1999), the discussion about a standard query language for XML was still ongoing. In this section, we will give an informal example of one specific query language for XML called **XML-QL** that has strong similarities to several query languages that have been developed in the database community (see Section 22.3.5).

Consider again the XML document shown in Figure 22.6. The following example query returns the last names of all authors, assuming that our XML document resides at the location `www.ourbookstore.com/books.xml`.

```
WHERE <BOOK>
        <NAME><LASTNAME> $l </LASTNAME></NAME>
      </BOOK> IN "www.ourbookstore.com/books.xml"
CONSTRUCT <RESULTNAME> $l </RESULTNAME>
```

This query extracts data from the XML document by specifying a pattern of markups. We are interested in data that is nested inside a `BOOK` element, a `NAME` element, and

a LASTNAME element. For each part of the XML document that matches the structure specified by the query, the variable l is bound to the contents of the element LASTNAME. To distinguish variable names from normal text, variable names are prefixed with a dollar sign $. If this query is applied to the sample data shown in Figure 22.6, the result would be the following XML document:

```
<RESULTNAME>Feynman</RESULTNAME>
<RESULTNAME>Narayan</RESULTNAME>
```

Selections are expressed by placing text in the content of an element. Also, the output of a query is not limited to a single element. We illustrate these two points in the next query. Assume that we want to find the last names and first names of all authors who wrote a book that was published in 1980. We can express this query as follows:

```
WHERE <BOOK> <NAME>
           <LASTNAME> $l </LASTNAME>
           <FIRSTNAME> $f </FIRSTNAME>
         </NAME>
         <PUBLISHED>1980</PUBLISHED>
       </BOOK> IN "www.ourbookstore.com/books.xml"
CONSTRUCT    <RESULTNAME>
               <FIRST>$f</FIRST><LAST>$l</LAST>
             </RESULTNAME>
```

The result of the above query is the following XML document:

```
<RESULTNAME><FIRST>Richard</FIRST><LAST>Feynman</LAST></RESULTNAME>
<RESULTNAME><FIRST>R.K.</FIRST><LAST>Narayan</LAST></RESULTNAME>
```

We conclude our discussion with a slightly more complicated example. Suppose that for each year we want to find the last names of authors who wrote a book published in that year. We group by PUBLISHED and the result contains a list of last names for each year:

```
WHERE <BOOK> $e <BOOK> IN "www.ourbookstore.com/books.xml",
      <AUTHOR>$n</AUTHOR>,
      <PUBLISHED>$p</PUBLISHED> IN $e
CONSTRUCT <RESULT><PUBLISHED> $p </PUBLISHED>
      WHERE <LASTNAME> $l </LASTNAME> IN $n
      CONSTRUCT <LASTNAME> $l </LASTNAME>
    </RESULT>
```

Using the XML document in Figure 22.6 as input, this query produces the following result:

Commercial database systems and XML: Many relational and object-relational database system vendors are currently looking into support for XML in their database engines. Several vendors of object-oriented database management systems already offer database engines that can store XML data whose contents can be accessed through graphical user interfaces, server-side Java extensions, or through XML-QL queries.

```
<RESULT> <PUBLISHED>1980</PUBLISHED>
  <LASTNAME>Feynman</LASTNAME>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
<RESULT> <PUBLISHED>1981</PUBLISHED>
  <LASTNAME>Narayan</LASTNAME>
</RESULT>
```

22.3.5 The Semistructured Data Model

Consider a set of documents on the Web that contain hyperlinks to other documents. These documents, although not completely unstructured, cannot be modeled naturally in the relational data model because the pattern of hyperlinks is not regular across documents. A bibliography file also has a certain degree of structure due to fields such as *author* and *title*, but is otherwise unstructured text. While some data is completely unstructured—for example video streams, audio streams, and image data—a lot of data is neither completely unstructured nor completely structured. We refer to data with partial structure as **semistructured data**. XML documents represent an important and growing source of semistructured data, and the theory of semistructured data models and queries has the potential to serve as the foundation for XML.

There are many reasons why data might be semistructured. First, the structure of data might be implicit, hidden, unknown, or the user might choose to ignore it. Second, consider the problem of integrating data from several heterogeneous sources where data exchange and transformation are important problems. We need a highly flexible data model to integrate data from all types of data sources including flat files and legacy systems; a structured data model is often too rigid. Third, we cannot query a structured database without knowing the schema, but sometimes we want to query the data without full knowledge of the schema. For example, we cannot express the query "Where in the database can we find the string Malgudi?" in a relational database system without knowing the schema.

All data models proposed for semistructured data represent the data as some kind of labeled graph. Nodes in the graph correspond to compound objects or atomic values,

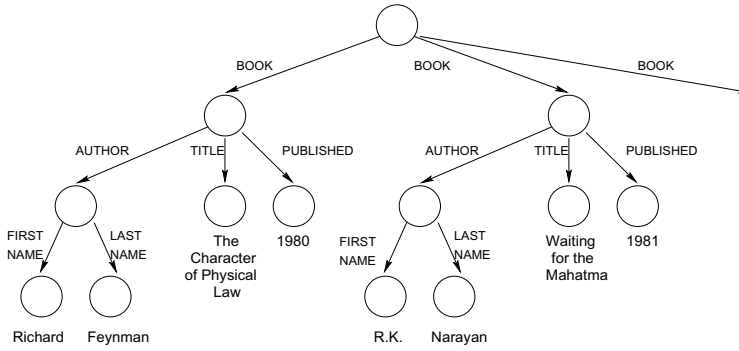

Figure 22.8 The Semistructured Data Model

and edges correspond to attributes. There is no separate schema and no auxiliary description; the data in the graph is self describing. For example, consider the graph shown in Figure 22.8, which represents part of the XML data from Figure 22.6. The root node of the graph represents the outermost element, `BOOKLIST`. The node has three outgoing edges that are labeled with the element name `BOOK`, since the list of books consists of three individual books.

We now discuss one of the proposed data models for semistructured data, called the **object exchange model (OEM)**. Each object is described by a triple consisting of a *label,* a *type,* and the *value* of the object. (An object in the object exchange model also has an object identifier, which is a unique identifier for the object. We omit object identifiers from our discussion here; the references at the end of the chapter provide pointers for the interested reader.) Since each object has a label that can be thought of as the column name in the relational model, and each object has a type that can be thought of as the column type in the relational model, the object exchange model is basically self-describing. Labels in the object exchange model should be as informative as possible, since they can serve two purposes: They can be used to identify an object as well as to convey the meaning of an object. For example, we can represent the last name of an author as follows:

⟨`lastName, string, "Feynman"`⟩

More complex objects are decomposed hierarchically into smaller objects. For example, an author name can contain a first name and a last name. This object is described as follows:

⟨`authorName, set,` $\{firstname_1, lastname_1\}$⟩
  $firstname_1$ `is` ⟨`firstName, string, "Richard"`⟩
  $lastname_1$ `is` ⟨`lastName, string, "Feynman"`⟩

As another example, an object representing a set of books is described as follows:

⟨`bookList, set,` $\{book_1, book_2, book_3\}$⟩
  $book_1$ `is` ⟨`book, set,` $\{author_1, title_1, published_1\}$⟩
  $book_2$ `is` ⟨`book, set,` $\{author_2, title_2, published_2\}$⟩
  $book_3$ `is` ⟨`book, set,` $\{author_3, title_3, published_3\}$⟩
    $author_3$ `is` ⟨`author, set,` $\{firstname_3, lastname_3\}$⟩
    $title_3$ `is` ⟨`title, string, "The English Teacher"`⟩
    $published_3$ `is` ⟨`published, integer, 1980`⟩

22.3.6 Implementation Issues for Semistructured Data

Database system support for semistructured data has been the focus of much research recently, and given the commercial success of XML, this emphasis is likely to continue. Semistructured data poses new challenges, since most of the traditional storage, indexing, and query processing strategies assume that the data adheres to a regular schema. For example, should we store semistructured data by mapping it into the relational model and then store the mapped data in a relational database system? Or does a storage subsystem specialized for semistructured data deliver better performance? How can we index semistructured data? Given a query language like XML-QL, what are suitable query processing strategies? Current research tries to find answers to these questions.

22.4 INDEXING FOR TEXT SEARCH

In this section, we assume that our database is a collection of documents and we call such a database a **text database**. For simplicity, we assume that the database contains exactly one relation and that the relation schema has exactly one field of type document. Thus, each record in the relation contains exactly one document. In practice, the relation schema would contain other fields such as the date of the creation of the document, a possible classification of the document, or a field with keywords describing the document. Text databases are used to store newspaper articles, legal papers, and other types of documents.

An important class of queries based on **keyword search** enables us to ask for all documents containing a given keyword. This is the most common kind of query on the Web today, and is supported by a number of search engines such as AltaVista and Lycos. Some systems maintain a list of synonyms for important words and return documents that contain a desired keyword or one of its synonyms; for example, a query asking for documents containing *car* will also retrieve documents containing *automobile*. A more complex query is "Find all documents that have *keyword1* `AND` *keyword2*." For such composite queries, constructed with `AND`, `OR`, and `NOT`, we can *rank* retrieved documents by the proximity of the query keywords in the document.

There are two common types of queries for text databases: boolean queries and ranked queries. In a **boolean query**, the user supplies a boolean expression of the following form, which is called **conjunctive normal form**:

$$(t_{11} \vee t_{12} \vee \ldots \vee t_{1i_1}) \wedge \ldots \wedge (t_{j1} \vee t_{12} \vee \ldots \vee t_{1i_j}),$$

where the t_{ij} are individual query terms or keywords. The query consists of j **conjuncts**, each of which consists of several **disjuncts**. In our query, the first conjunct is the expression $(t_{11} \vee t_{12} \vee \ldots \vee t_{1i_1})$; it consists of i_1 disjuncts. Queries in conjunctive normal form have a natural interpretation. The result of the query are documents that involve several concepts. Each conjunct corresponds to one concept, and the different words within each conjunct correspond to different terms for the same concept.

Ranked queries are structurally very similar. In a **ranked query** the user also specifies a list of words, but the result of the query is a list of documents ranked by their relevance to the list of user terms. How to define when and how relevant a document is to a set of user terms is a difficult problem. Algorithms to evaluate such queries belong to the field of **information retrieval**, which is closely related to database management. Information retrieval systems, like database systems, have the goal of enabling users to query a large volume of data, but the focus has been on large collections of unstructured documents. Updates, concurrency control, and recovery have traditionally not been addressed in information retrieval systems because the data in typical applications is largely static.

The criteria used to evaluate such information retrieval systems include **precision**, which is the percentage of retrieved documents that are relevant to the query, and **recall**, which is the percentage of relevant documents in the database that are retrieved in response to a query.

The advent of the Web has given fresh impetus to information retrieval because millions of documents are now available to users and searching for desired information is a fundamental operation; without good search mechanisms, users would be overwhelmed. An index for an information retrieval system essentially contains $\langle$*keyword,documentid*$\rangle$ pairs, possibly with additional fields such as the number of times a keyword appears in a document; a Web search engine creates a centralized index for documents that are stored at several sites.

In the rest of this section, we concentrate on boolean queries. We introduce two index schemas that support the evaluation of boolean queries efficiently. The *inverted file* index discussed in Section 22.4.1 is widely used due to its simplicity and good performance. Its main disadvantage is that it imposes a significant space overhead: The size can be up to 300 percent the size of the original file. The *signature file* index discussed in Section 22.4.2 has a small space overhead and offers a quick filter that eliminates most nonqualifying documents. However, it scales less well to larger

Rid	Document	Signature
1	agent James Bond	1100
2	agent mobile computer	1101
3	James Madison movie	1011
4	James Bond movie	1110

Word	Inverted list	Hash
agent	$\langle 1, 2 \rangle$	1000
Bond	$\langle 1, 4 \rangle$	0100
computer	$\langle 2 \rangle$	0100
James	$\langle 1, 3, 4 \rangle$	1000
Madison	$\langle 3 \rangle$	0001
mobile	$\langle 2 \rangle$	0001
movie	$\langle 3, 4 \rangle$	0010

Figure 22.9 A Text Database with Four Records and Indexes

database sizes because the index has to be sequentially scanned. We discuss evaluation of ranked queries in Section 22.5.

We assume that slightly different words that have the same root have been **stemmed**, or analyzed for the common root, during index creation. For example, we assume that the result of a query on 'index' also contains documents that include the terms 'indexes' and 'indexing.' Whether and how to stem is application dependent, and we will not discuss the details.

As a running example, we assume that we have the four documents shown in Figure 22.9. For simplicity, we assume that the record identifiers of the four documents are the numbers one to four. Usually the record identifiers are not physical addresses on the disk, but rather entries in an **address table**. An address table is an array that maps the logical record identifiers, as shown in Figure 22.9, to physical record addresses on disk.

22.4.1 Inverted Files

An **inverted file** is an index structure that enables fast retrieval of all documents that contain a query term. For each term, the index maintains an ordered list (called the **inverted list**) of document identifiers that contain the indexed term. For example, consider the text database shown in Figure 22.9. The query term 'James' has the inverted list of record identifiers $\langle 1, 3, 4 \rangle$ and the query term 'movie' has the list $\langle 3, 4 \rangle$. Figure 22.9 shows the inverted lists of all query terms.

In order to quickly find the inverted list for a query term, all possible query terms are organized in a second index structure such as a B+ tree or a hash index. To avoid any confusion, we will call the second index that allows fast retrieval of the inverted list for a query term the **vocabulary index**. The vocabulary index contains each possible query term and a pointer to its inverted list.

A query containing a single term is evaluated by first traversing the vocabulary index to the leaf node entry with the address of the inverted list for the term. Then the inverted list is retrieved, the rids are mapped to physical document addresses, and the corresponding documents are retrieved. A query with a conjunction of several terms is evaluated by retrieving the inverted lists of the query terms one at a time and intersecting them. In order to minimize memory usage, the inverted lists should be retrieved in order of increasing length. A query with a disjunction of several terms is evaluated by merging all relevant inverted lists.

Consider again the example text database shown in Figure 22.9. To evaluate the query 'James', we probe the vocabulary index to find the inverted list for 'James', fetch the inverted list from disk and then retrieve document one. To evaluate the query 'James' `AND` 'Bond', we retrieve the inverted list for the term 'Bond' and intersect it with the inverted list for the term 'James.' (The inverted list of the term 'Bond' has length two, whereas the inverted list of the term 'James' has length three.) The result of the intersection of the list $\langle 1, 4 \rangle$ with the list $\langle 1, 3, 4 \rangle$ is the list $\langle 1, 4 \rangle$ and the first and fourth document are retrieved. To evaluate the query 'James' `OR` 'Bond,' we retrieve the two inverted lists in any order and merge the results.

22.4.2 Signature Files

A **signature file** is another index structure for text database systems that supports efficient evaluation of boolean queries. A signature file contains an index record for each document in the database. This index record is called the **signature** of the document. Each signature has a fixed size of b bits; b is called the **signature width**. How do we decide which bits to set for a document? The bits that are set depend on the words that appear in the document. We map words to bits by applying a hash function to each word in the document and we set the bits that appear in the result of the hash function. Note that unless we have a bit for each possible word in the vocabulary, the same bit could be set twice by different words because the hash function maps both words to the same bit. We say that a signature S_1 matches another signature S_2 if all the bits that are set in signature S_2 are also set in signature S_1. If signature S_1 matches signature S_2, then signature S_1 has at least as many bits set as signature S_2.

For a query consisting of a conjunction of terms, we first generate the query signature by applying the hash function to each word in the query. We then scan the signature file and retrieve all documents whose signatures match the query signature, because every such document is a potential result to the query. Since the signature does not uniquely identify the words that a document contains, we have to retrieve each potential match and check whether the document actually contains the query terms. A document whose signature matches the query signature but that does not contain all terms in the query is called a **false positive**. A false positive is an expensive mistake since the

document has to be retrieved from disk, parsed, stemmed, and checked to determine whether it contains the query terms.

For a query consisting of a disjunction of terms, we generate a list of query signatures, one for each term in the query. The query is evaluated by scanning the signature file to find documents whose signatures match any signature in the list of query signatures.

Note that for each query we have to scan the complete signature file, and there are as many records in the signature file as there are documents in the database. To reduce the amount of data that has to be retrieved for each query, we can vertically partition a signature file into a set of **bit slices**, and we call such an index a **bit-sliced signature file**. The length of each bit slice is still equal to the number of documents in the database, but for a query with q bits set in the query signature we need only to retrieve q bit slices.

As an example, consider the text database shown in Figure 22.9 with a signature file of width 4. The bits set by the hashed values of all query terms are shown in the figure. To evaluate the query 'James,' we first compute the hash value of the term which is 1000. Then we scan the signature file and find matching index records. As we can see from Figure 22.9, the signatures of all records have the first bit set. We retrieve all documents and check for false positives; the only false positive for this query is document with rid 2. (Unfortunately, the hashed value of the term 'agent' also sets the very first bit in the signature.) Consider the query 'James' `AND` 'Bond.' The query signature is 1100 and three document signatures match the query signature. Again, we retrieve one false positive. As another example of a conjunctive query, consider the query 'movie' `AND` 'Madison.' The query signature is 0011, and only one document signature matches the query signature. No false positives are retrieved. The reader is invited to construct a bit-sliced signature file and to evaluate the example queries in this paragraph using the bit slices.

22.5 RANKED KEYWORD SEARCHES ON THE WEB

The World Wide Web contains a mind-boggling amount of information. Finding Web pages that are relevant to a user query can be more difficult than finding a needle in a haystack. The variety of pages in terms of structure, content, authorship, quality, and validity of the data makes it difficult if not impossible to apply standard retrieval techniques.

For example, a boolean text search as discussed in Section 22.4 is not sufficient because the result for a query with a single term could consist of links to thousands, if not millions of pages, and we rarely have the time to browse manually through all of them. Even if we pose a more sophisticated query using conjunction and disjunction of terms the number of Web pages returned is still several hundreds for any topic of reasonable

breadth. Thus, querying effectively using a boolean keyword search requires expert users who can carefully combine terms specifying a very narrowly defined subject.

One natural solution to the excessive number of answers returned by boolean keyword searches is to take the output of the boolean text query and somehow process this set further to find the most relevant pages. For abstract concepts, however, often the most relevant pages do not contain the search terms at all and are therefore not returned by a boolean keyword search! For example, consider the query term 'Web browser.' A boolean text query using the terms does not return the relevant pages of Netscape Corporation or Microsoft, because these pages do not contain the term 'Web browser' at all. Similarly, the home page of Yahoo does not contain the term 'search engine.' The problem is that relevant sites do not necessarily describe their contents in a way that is useful for boolean text queries.

Until now, we only considered information within a single Web page to estimate its relevance to a query. But Web pages are connected through hyperlinks, and it is quite likely that there is a Web page containing the term 'search engine' that has a link to Yahoo's home page. Can we use the information hidden in such links?

In our search for relevant pages, we distinguish between two types of pages: *authorities* and *hubs*. An **authority** is a page that is very relevant to a certain topic and that is recognized by other pages as authoritative on the subject. These other pages, called hubs, usually have a significant number of hyperlinks to authorities, although they themselves are not very well known and do not necessarily carry a lot of content relevant to the given query. **Hub** pages could be compilations of resources about a topic on a site for professionals, lists of recommended sites for the hobbies of an individual user, or even a part of the bookmarks of an individual user that are relevant to one of the user's interests; their main property is that they have many outgoing links to relevant pages. Good hub pages are often not well known and there may be few links pointing to a good hub. In contrast, good authorities are 'endorsed' by many good hubs and thus have many links from good hub pages.

We will use this symbiotic relationship between hubs and authorities in the HITS algorithm, a link-based search algorithm that discovers high-quality pages that are relevant to a user's query terms.

22.5.1 An Algorithm for Ranking Web Pages

In this section we will discuss HITS, an algorithm that finds good authorities and hubs and returns them as the result of a user query. We view the World Wide Web as a directed graph. Each Web page represents a node in the graph, and a hyperlink from page A to page B is represented as an edge between the two corresponding nodes.

Assume that we are given a user query with several terms. The algorithm proceeds in two steps. In the first step, the *sampling step*, we collect a set of pages called the **base set**. The base set most likely includes very relevant pages to the user's query, but the base set can still be quite large. In the second step, the *iteration step*, we find good authorities and good hubs among the pages in the base set.

The sampling step retrieves a set of Web pages that contain the query terms, using some traditional technique. For example, we can evaluate the query as a boolean keyword search and retrieve all Web pages that contain the query terms. We call the resulting set of pages the **root set**. The root set might not contain all relevant pages because some authoritative pages might not include the user query words. But we expect that at least some of the pages in the root set contain hyperlinks to the most relevant authoritative pages or that some authoritative pages link to pages in the root set. This motivates our notion of a **link page**. We call a page a link page if it has a hyperlink to some page in the root set or if a page in the root set has a hyperlink to it. In order not to miss potentially relevant pages, we augment the root set by all link pages and we call the resulting set of pages the **base set**. Thus, the base set includes all root pages and all link pages; we will refer to a Web page in the base set as a **base page**.

Our goal in the second step of the algorithm is to find out which base pages are good hubs and good authorities and to return the best authorities and hubs as the answers to the query. To quantify the quality of a base page as a hub and as an authority, we associate with each base page in the base set a **hub weight** and an **authority weight**. The hub weight of the page indicates the quality of the page as a hub, and the authority weight of the page indicates the quality of the page as an authority. We compute the weights of each page according to the intuition that a page is a good authority if many good hubs have hyperlinks to it, and that a page is a good hub if it has many outgoing hyperlinks to good authorities. Since we do not have any a priori knowledge about which pages are good hubs and authorities, we initialize all weights to one. We then update the authority and hub weights of base pages iteratively as described below.

Consider a base page p with hub weight h_p and with authority weight a_p. In one iteration, we update a_p to be the sum of the hub weights of all pages that have a hyperlink to p. Formally:

$$a_p = \sum_{\text{All base pages } q \text{ that have a link to } p} h_q$$

Analogously, we update h_p to be the sum of the weights of all pages that p points to:

$$h_p = \sum_{\text{All base pages } q \text{ such that } p \text{ has a link to } q} a_q$$

Computing hub and authority weights: We can use matrix notation to write the updates for all hub and authority weights in one step. Assume that we number all pages in the base set $\{1, 2, ..., n\}$. The adjacency matrix B of the base set is an $n \times n$ matrix whose entries are either 0 or 1. The matrix entry (i, j) is set to 1 if page i has a hyperlink to page j; it is set to 0 otherwise. We can also write the hub weights h and authority weights a in vector notation: $h = \langle h_1, \ldots, h_n \rangle$ and $a = \langle a_1, \ldots, a_n \rangle$. We can now rewrite our update rules as follows:

$$h = B \cdot a, \quad \text{and} \quad a = B^T \cdot h \ .$$

Unfolding this equation once, corresponding to the first iteration, we obtain:

$$h = BB^T h = (BB^T)h, \quad \text{and} \quad a = B^T Ba = (B^T B)a \ .$$

After the second iteration, we arrive at:

$$h = (BB^T)^2 h, \quad \text{and} \quad a = (B^T B)^2 a \ .$$

Results from linear algebra tell us that the sequence of iterations for the hub (resp. authority) weights converges to the principal eigenvectors of BB^T (resp. $B^T B$) if we normalize the weights before each iteration so that the sum of the squares of all weights is always $2 \cdot n$. Furthermore, results from linear algebra tell us that this convergence is independent of the choice of initial weights, as long as the initial weights are positive. Thus, our rather arbitrary choice of initial weights—we initialized all hub and authority weights to 1—does not change the outcome of the algorithm.

Comparing the algorithm with the other approaches to querying text that we discussed in this chapter, we note that the iteration step of the HITS algorithm—the distribution of the weights—does not take into account the words on the base pages. In the iteration step, we are only concerned about the relationship between the base pages as represented by hyperlinks.

The HITS algorithm often produces very good results. For example, the five highest ranked authorities for the query 'search engines' are the following Web pages:

```
http://www.yahoo.com/
http://www.excite.com/
http://www.mckinley.com/
http://www.lycos.com/
http://www.altavista.digital.com/
```

The three highest ranked authorities for the query containing the single keyword 'Gates' are the following Web pages:

```
http://www.roadahead.com/
http://www.microsoft.com/
http://www.microsoft.com/corpinfo/bill-g.htm
```

22.6 POINTS TO REVIEW

- Files on the World Wide Web are identified through *universal resource locators (URLs)*. A *Web browser* takes a URL, goes to the site containing the file, and asks the *Web server* at that site for the file. It then displays the file appropriately, taking into account the type of file and the formatting instructions that it contains. The browser calls application programs to handle certain types of files, e.g., it calls Microsoft Word to handle Word documents (which are identified through a `.doc` file name extension). *HTML* is a simple *markup language* used to describe a document. Audio, video, and even Java programs can be included in HTML documents.

  Increasingly, data accessed through the Web is stored in DBMSs. A Web server can access data in a DBMS to construct a page requested by a Web browser. **(Section 22.1)**

- A Web server often has to execute a program at its site in order to satisfy a request from a Web browser (which is usually executing at a different site). For example, it may have to access data in a DBMS. There are two ways for a Web server to execute a program: It can create a new process and communicate with it using the *CGI* protocol, or it can create a new thread (or invoke an existing thread) for a *Java servlet*. The second approach avoids the overhead of creating a new process for each request. An *application server* manages several threads and provides other functionality to facilitate executing programs at the Web server's site. The additional functionality includes security, session management, and coordination of access to multiple data sources. *JavaBeans* and *Java Server Pages* are Java-based technologies that assist in creating and managing programs designed to be invoked by a Web server. **(Section 22.2)**

- *XML* is an emerging document description standard that allows us to describe the *content* and *structure* of a document in addition to giving display directives. It is based upon HTML and *SGML*, which is a powerful document description standard that is widely used. XML is designed to be simple enough to permit easy manipulation of XML documents, in contrast to SGML, while allowing users to develop their own document descriptions, unlike HTML. In particular, a DTD is a document description that is independent of the contents of a document, just like a relational database schema is a description of a database that is independent of the actual database instance. The development of DTDs for different application domains offers the potential that documents in these domains can be freely exchanged and uniformly interpreted according to standard, agreed-upon DTD descriptions. XML documents have less rigid structure than a relational database

and are said to be *semistructured*. Nonetheless, there is sufficient structure to permit many useful queries, and query languages are being developed for XML data. **(Section 22.3)**

- The proliferation of text data on the Web has brought renewed attention to information retrieval techniques for searching text. Two broad classes of search are *boolean queries* and *ranked queries*. *Boolean queries* ask for documents containing a specified boolean combination of keywords. *Ranked queries* ask for documents that are most relevant to a given list of keywords; the quality of answers is evaluated using *precision* (the percentage of retrieved documents that are relevant to the query) and *recall* (the percentage of relevant documents in the database that are retrieved) as metrics.

  Inverted files and *signature files* are two indexing techniques that support boolean queries. Inverted files are widely used and perform well, but have a high space overhead. Signature files address the space problem associated with inverted files but must be sequentially scanned. **(Section 22.4)**

- Handling ranked queries on the Web is a difficult problem. The HITS algorithm uses a combination of boolean queries and analysis of links to a page from other Web sites to evaluate ranked queries. The intuition is to find authoritative sources for the concepts listed in the query. An authoritative source is likely to be frequently cited. A good source of citations is likely to cite several good authorities. These observations can be used to assign weights to sites and identify which sites are good authorities and hubs for a given query. **(Section 22.5)**

EXERCISES

Exercise 22.1 Briefly answer the following questions.

1. Define the following terms and describe what they are used for: HTML, URL, CGI, server-side processing, Java Servlet, JavaBean, Java server page, HTML template, CCS, XML, DTD, XSL, semistructured data, inverted file, signature file.
2. What is CGI? What are the disadvantages of an architecture using CGI scripts?
3. What is the difference between a Web server and an application server? What funcionality do typical application servers provide?
4. When is an XML document well-formed? When is an XML document valid?

Exercise 22.2 Consider our bookstore example again. Assume that customers also want to search books by title.

1. Extend the HTML document shown in Figure 22.2 by another form that allows users to input the title of a book.
2. Write a Perl script similar to the Perl script shown in Figure 22.3 that generates dynamically an HTML page with all books that have the user-specified title.

Exercise 22.3 Consider the following description of items shown in the Eggface computer mail-order catalog.

"Eggface sells hardware and software. We sell the new Palm Pilot V for $400; its part number is 345. We also sell the IBM ThinkPad 570 for only $1999; choose part number 3784. We sell both business and entertainment software. Microsoft Office 2000 has just arrived and you can purchase the Standard Edition for only $140, part number 974. The new desktop publishing software from Adobe called InDesign is here for only $200, part 664. We carry the newest games from Blizzard software. You can start playing Diablo II for only $30, part number 12, and you can purchase Starcraft for only $10, part number 812."

1. Design an HTML document that depicts the items offered by Eggface.
2. Create a well-formed XML document that describes the contents of the Eggface catalog.
3. Create a DTD for your XML document and make sure that the document you created in the last question is valid with respect to this DTD.
4. Write an XML-QL query that lists all software items in the catalog.
5. Write an XML-QL query that lists the prices of all hardware items in the catalog.
6. Depict the catalog data in the semistructured data model as shown in Figure 22.8.

Exercise 22.4 A university database contains information about professors and the courses they teach. The university has decided to publish this information on the Web and you are in charge of the execution. You are given the following information about the contents of the database:

In the fall semester 1999, the course 'Introduction to Database Management Systems' was taught by Professor Ioannidis. The course took place Mondays and Wednesdays from 9–10 a.m. in room 101. The discussion section was held on Fridays from 9–10 a.m. Also in the fall semester 1999, the course 'Advanced Database Management Systems' was taught by Professor Carey. Thirty five students took that course which was held in room 110 Tuesdays and Thursdays from 1–2 p.m. In the spring semester 1999, the course 'Introduction to Database Management Systems' was taught by U.N. Owen on Tuesdays and Thursdays from 3–4 p.m. in room 110. Sixty three students were enrolled; the discussion section was on Thursdays from 4–5 p.m. The other course taught in the spring semester was 'Advanced Database Management Systems' by Professor Ioannidis, Monday, Wednesday, and Friday from 8–9 a.m.

1. Create a well-formed XML document that contains the university database.
2. Create a DTD for your XML document. Make sure that the XML document is valid with respect to this DTD.
3. Write an XML-QL query that lists the name of all professors.
4. Describe the information in a different XML document—a document that has a different structure. Create a corresponding DTD and make sure that the document is valid. Reformulate your XML-QL query that finds the names of all professors to work with the new DTD.

Exercise 22.5 Consider the database of the FamilyWear clothes manufacturer. FamilyWear produces three types of clothes: women's clothes, men's clothes, and children's clothes. Men can choose between polo shirts and T-shirts. Each polo shirt has a list of available colors, sizes, and a uniform price. Each T-shirt has a price, a list of available colors, and a list of available sizes. Women have the same choice of polo shirts and T-shirts as men. In addition women can choose between three types of jeans: slim fit, easy fit, and relaxed fit jeans. Each pair of jeans has a list of possible waist sizes and possible lengths. The price of a pair of jeans only depends on its type. Children can choose between T-shirts and baseball caps. Each T-shirt has a price, a list of available colors, and a list of available patterns. T-shirts for children all have the same size. Baseball caps come in three different sizes: small, medium, and large. Each item has an optional sales price that is offered on special occasions.

Design an XML DTD for FamilyWear so that FamilyWear can publish its catalog on the Web.

Exercise 22.6 Assume you are given a document database that contains six documents. After stemming, the documents contain the following terms:

Document	Terms
1	car manufacturer Honda auto
2	auto computer navigation
3	Honda navigation
4	manufacturer computer IBM
5	IBM personal computer
6	car Beetle VW

Answer the following questions.

1. Discuss the advantages and disadvantages of inverted files versus signature files.
2. Show the result of creating an inverted file on the documents.
3. Show the result of creating a signature file with a width of 5 bits. Construct your own hashing function that maps terms to bit positions.
4. Evaluate the following queries using the inverted file and the signature file that you created: 'car', 'IBM' `AND` 'computer', 'IBM' `AND` 'car', 'IBM' `OR` 'auto', and 'IBM' `AND` 'computer' `AND` 'manufacturer'.
5. Assume that the query load against the document database consists of exactly the queries that were stated in the previous question. Also assume that each of these queries is evaluated exactly once.
   (a) Design a signature file with a width of 3 bits and design a hashing function that minimizes the overall number of false positives retrieved when evaluating the
   (b) Design a signature file with a width of 6 bits and a hashing function that minimizes the overall number of false positives.
   (c) Assume you want to construct a signature file. What is the smallest signature width that allows you to evaluate all queries without retrieving any false positives?

Exercise 22.7 Assume that the base set of the HITS algorithm consists of the set of Web pages displayed in the following table. An entry should be interpreted as follows: Web page 1 has hyperlinks to pages 5 and 6.

Web page	Pages that this page has links to
1	5, 6, 7
2	5, 7
3	6, 8
4	
5	1, 2
6	1, 3
7	1, 2
8	4

Run five iterations of the HITS algorithm and find the highest ranked authority and the highest ranked hub.

BIBLIOGRAPHIC NOTES

The latest version of the standards mentioned in this chapter can be found from the Web pages of the World Wide Web Consortium (`www.w3.org`). Its Web site contains links to information about HTML, cascading style sheets, XML, XSL, and much more. The book by Hall is a general introduction to Web programming technologies [302]; a good starting point on the Web is `www.Webdeveloper.com`. There are many introductory books on CGI programming, for example [176, 166]. The JavaSoft (`java.sun.com`) home page is a good starting point for JavaBeans, Servlets, JSP, and all other Java-related technologies. The book by Hunter [333] is a good introduction to Java Servlets. Microsoft supports Active Server Pages (ASP), a comparable technology to JSP. More information about ASP can be found on the Microsoft Developer's Network homepage (`msdn.microsoft.com`).

There are excellent Web sites devoted to the advancement of XML, for example `www.xml.com` and `www.ibm.com/xml`, that also contain a plethora of links with information about the other standards. There are good introductory books on many different aspects of XML, for example [164, 135, 520, 411, 321, 271]. Information about UNICODE can be found on its home page `http://www.unicode.org`.

There is a lot of research on semistructured data in the database community. The Tsimmis data integration system uses a semistructured data model to cope with possible heterogeneity of data sources [509, 508]. Several new query languages for semistructured data have been developed: LOREL [525], UnQL [106], StruQL [233], and WebSQL [458]. LORE is a database management system designed for semistructured data [450]. Query optimization for semistructured data is addressed in [5, 272]. Work on describing the structure of semistructured databases can be found in [490, 272].

There has been a lot of work on using semistructured data models for Web data and several Web query systems have been developed: WebSQL [458], W3QS [384], WebLog [399], WebOQL [32], STRUDEL [232], ARANEUS [39], and FLORID [319]. [237] is a good overview of database research in the context of the Web.

Introductory reading material on information retrieval includes the standard textbooks by Salton and McGill [562] and by van Rijsbergen [661]. Collections of articles for the more

advanced reader have been edited by Jones and Willett [350] and by Frakes and Baeza-Yates [239]. Querying text repositories has been studied extensively in information retrieval; see [545] for a recent survey. Faloutsos overviews indexing methods for text databases [219]. Inverted files are discussed in [469] and signature files are discussed in [221]. Zobel, Moffat, and Ramamohanarao give a comparison of inverted files and signature files [703]. Other aspects of indexing for text databases are addressed in [704]. The book by Witten, Moffat, and Bell has a lot of material on compression techniques for document databases [685].

The number of citation counts as a measure of scientific impact has first been studied by Garfield [262]; see also [670]. Usage of hypertextual information to improve the quality of search engines has been proposed by Spertus [610] and by Weiss et al. [676]. The HITS algorithm was developed by Jon Kleinberg [376]. Concurrently, Brin and Page developed the Pagerank algorithm, which also takes hyperlinks between pages into account [99]. The discovery of structure in the World Wide Web is currently a very active area of research; see for example the work by Gibson et al. [268].