C H A P T E R 2 1

Application Development and Administration

Practically all use of databases occurs from within application programs. Correspondingly, almost all user interaction with databases is indirect, via application programs. Not surprisingly, therefore, database systems have long supported tools such as form and GUI builders, which help in rapid development of applications that interface with users. In recent years, the Web has become the most widely used user interface to databases.

Once an application has been built, it is often found to run slower than the designers wanted, or to handle fewer transactions per second than they required. Applications can be made to run significantly faster by performance tuning, which consists of finding and eliminating bottlenecks and adding appropriate hardware such as memory or disks. Benchmarks help to characterize the performance of database systems.

Standards are very important for application development, especially in the age of the internet, since applications need to communicate with each other to perform useful tasks. A variety of standards have been proposed that affect database application development.

Electronic commerce is becoming an integral part of how we purchase goods and services and databases play an important role in that domain.

Legacy systems are systems based on older-generation technology. They are often at the core of organizations, and run mission-critical applications. We outline issues in interfacing with legacy systems, and how they can be replaced by other systems.

21.1 Web Interfaces to Databases

The **World Wide Web** (**Web**, for short), is a distributed information system based on hypertext. Web interfaces to databases have become very important. After outlining several reasons for interfacing databases with the Web (Section 21.1.1), we provide an overview of Web technology (Section 21.1.2) and then study Web servers (Section 21.1.3) and outline some state-of-the art techniques for building Web interfaces to databases, using servlets and server-side scripting languages (Sections 21.1.4 and 21.1.5). We describe techniques for improving performance in Section 21.1.6.

21.1.1 Motivation

The Web has become important as a front end to databases for several reasons: Web browsers provide a *universal front end* to information supplied by back ends located anywhere in the world. The front end can run on any computer system, and there is no need for a user to download any special-purpose software to access information. Further, today, almost everyone who can afford it has access to the Web.

With the growth of information services and electronic commerce on the Web, databases used for information services, decision support, and transaction processing must be linked with the Web. The HTML forms interface is convenient for transaction processing. The user can fill in details in an order form, then click a submit button to send a message to the server. The server executes an application program corresponding to the order form, and this action in turn executes transactions on a database at the server site. The server formats the results of the transaction and sends them back to the user.

Another reason for interfacing databases to the Web is that presenting only static (fixed) documents on a Web site has some limitations, even when the user is not doing any querying or transaction processing:

- Fixed Web documents do not allow the display to be tailored to the user. For instance, a newspaper may want to tailor its display on a per-user basis, to give prominence to news articles that are likely to be of interest to the user.
- When the company data are updated, the Web documents become obsolete if they are not updated simultaneously. The problem becomes more acute if multiple Web documents replicate important data, and all must be updated.

We can fix these problems by generating Web documents dynamically from a database. When a document is requested, a program gets executed at the server site, which in turn runs queries on a database, and generates the requested document on the basis of the query results. Whenever relevant data in the database are updated, the generated documents will automatically become up-to-date. The generated document can also be tailored to the user on the basis of user information stored in the database.

Web interfaces provide attractive benefits even for database applications that are used only with a single organization. The **HyperText Markup Language (HTML)** standard allows text to be neatly formatted, with important information highlighted. **Hyperlinks**, which are links to other documents, can be associated with regions of the displayed data. Clicking on a hyperlink fetches and displays the linked document. Hyperlinks are very useful for browsing data, permitting users to get more details of parts of the data as desired.

Finally, browsers today can fetch programs along with HTML documents, and run the programs on the browser, in safe mode—that is, without damaging data on the user's computer. Programs can be written in client-side scripting languages, such as Javascript, or can be "applets" written in the Java language. These programs permit the construction of sophisticated user interfaces, beyond what is possible with just HTML, interfaces that can be used without downloading and installing any software. Thus, Web interfaces are powerful and visually attractive, and are likely to eclipse special-purpose interfaces for all except a small class of users.

21.1.2 Web Fundamentals

Here we review some of the fundamental technology behind the World Wide Web, for readers who are not familiar with it.

21.1.2.1 Uniform Resource Locators

A **uniform resource locator** (**URL**) is a globally unique name for each document that can be accessed on the Web. An example of a URL is

http://www.bell-labs.com/topic/book/db-book

The first part of the URL indicates how the document is to be accessed: "http" indicates that the document is to be accessed by the HyperText Transfer Protocol, which is a protocol for transferring HTML documents. The second part gives the unique name of a machine that has a Web server. The rest of the URL is the path name of the file on the machine, or other unique identifier of the document within the machine.

Much data on the Web is dynamically generated. A URL can contain the identifier of a program located on the Web server machine, as well as arguments to be given to the program. An example of such a URL is

http://www.google.com/search?q=silberschatz

which says that the program search on the server www.google.com should be executed with the argument q=silberschatz. The program executes, using the given arguments, and returns an HTML document, which is then sent to the front end.

21.1.2.2 HyperText Markup Language

Figure 21.1 is an example of the source of an HTML document. Figure 21.2 shows the displayed image that this document creates.

The figures show how HTML can display a table and a simple form that allows users to select the type (account or loan) from a menu and to input a number in a text box. HTML also supports several other input types. Clicking on the submit button causes the program BankQuery (specified in the form action field) to be executed with the user-provided values for the arguments type and number (specified in the select and input fields). The program generates an HTML document, which is then sent back and displayed to the user; we will see how to construct such programs in Sections 21.1.3, 21.1.4, and 21.1.5.

HTML supports *stylesheets*, which can alter the default definitions of how an HTML formatting construct is displayed, as well as other display attributes such as background color of the page. The *cascading stylesheet (css)* standard allows the same

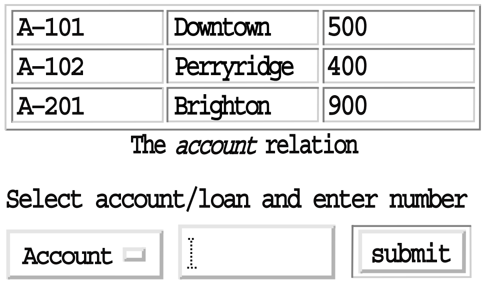
```
<html>
<body>
<table BORDER COLS=3>
<tr> <td>A-101</td> <td>Downtown</td> <td>500</td> </tr>
<tr> <td>A-102</td> <td>Perryridge</td> <td>400</td> </tr>
<tr> <td>A-201</td> <td>Brighton   </td> <td>900</td> </tr>
</table>
<center> The <i>account</i> relation </center>

<form action="BankQuery" method=get>
Select account/loan and enter number <br>
<select name="type">
      <option value="account" selected>Account
      <option value="loan"> Loan
</select>
<input type=text size=5 name="number">
<input type=submit value="submit">

</body>
</html>
```

Figure 21.1 An HTML source text.

stylesheet to be used for multiple HTML documents, giving a uniform look to all the pages on a Web site.

21.1.2.3 Client-Side Scripting and Applets

Embedding of program code in documents allows Web pages to be **active**, carrying out activities such as animation by executing programs at the local site, rather than just presenting passive text and graphics. The primary use of such programs is flexible interaction with the user, beyond the limited interaction power provided by HTML and HTML forms. Further, executing programs at the client site speeds up



A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900

The *account* relation

Select account/loan and enter number

Figure 21.2 Display of HTML source from Figure 21.1.

interaction greatly, compared to every interaction being sent to a server site for processing.

A danger in supporting such programs is that, if the design of the system is done carelessly, program code embedded in a Web page (or equivalently, in an e-mail message) can perform malicious actions on the user's computer. The malicious actions could range from reading private information, to deleting or modifying information on the computer, up to taking control of the computer and propagating the code to other computers (through e-mail, for example). A number of e-mail viruses have spread widely in recent years in this way.

The *Java* language became very popular because it provides a safe mode for executing programs on user's computers. Java code can be compiled into platform-independent "byte-code" that can be executed on any browser that supports Java. Unlike local programs, Java programs (applets) downloaded as part of a Web page have no authority to perform any actions that could be destructive. They are permitted to display data on the screen, or to make a network connection to the server from which the Web page was downloaded, in order to fetch more information. However, they are not permitted to access local files, to execute any system programs, or to make network connections to any other computers.

While Java is a full-fledged programming language, there are simpler languages, called **scripting languages**, that can enrich user interaction, while providing the same protection as Java. These languages provide constructs that can be embedded with an HTML document. **Client-side scripting languages** are languages designed to be executed on the client's Web browser. Of these, the *Javascript* language is by far the most widely used. There are also special-purpose scripting languages for specialized tasks such as animation (for example, Macromedia Flash and Shockwave), and three-dimensional modeling (Virtual Reality Markup Language (VRML)). Scripting languages can also be used on the server side, as we shall see.

21.1.3 Web Servers and Sessions

A **Web server** is a program running on the server machine, which accepts requests from a Web browser and sends back results in the form of HTML documents. The browser and Web server communicate by a protocol called the **HyperText Transfer Protocol (HTTP)**. HTTP provides powerful features, beyond the simple transfer of documents. The most important feature is the ability to execute programs, with arguments supplied by the user, and deliver the results back as an HTML document.

As a result, a Web server can easily act as an intermediary to provide access to a variety of information services. A new service can be created by creating and installing an application program that provides the service. The **common gateway interface (CGI)** standard defines how the Web server communicates with application programs. The application program typically communicates with a database server, through ODBC, JDBC, or other protocols, in order to get or store data.

Figure 21.3 shows a Web service using a three-tier architecture, with a Web server, an application server, and a database server. Using multiple levels of servers increases system overhead; the CGI interface starts a new process to service each request, which results in even greater overhead.

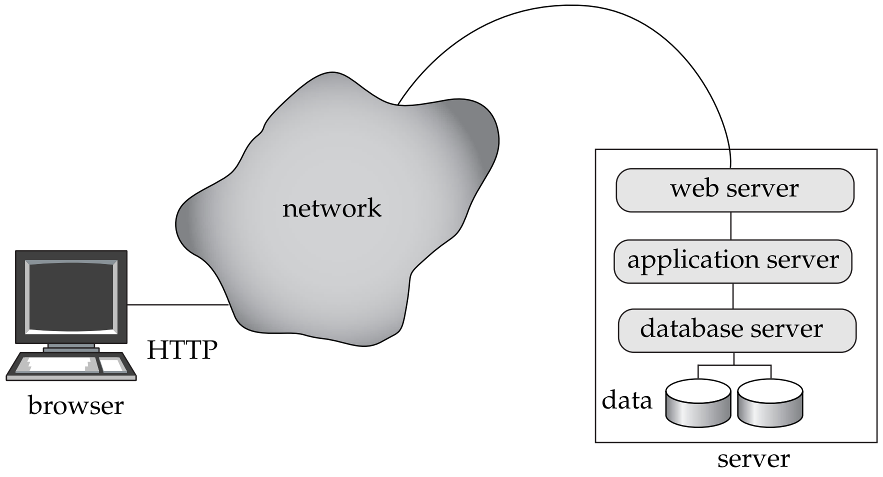


Figure 21.3 Three-tier Web architecture.

Most Web services today therefore use a two-tier Web architecture, where the application program runs within the Web server, as in Figure 21.4. We study systems based on the two-tier architecture in more detail in subsequent sections.

Be aware that there is no continuous connection between the client and the server. In contrast, when a user logs on to a computer, or connects to an ODBC or JDBC server, a session is created, and session information is retained at the server and the client until the session is terminated—information such as whether the user was authenticated using a password and what session options the user set. The reason that HTTP is **connectionless** is that most computers have limits on the number of simultaneous connections they can accommodate, and if a large number of sites on the Web open connections, this limit would be exceeded, denying service to further users. With a connectionless service, the connection is broken as soon as a request is satisfied, leaving connections available for other requests.

Most information services need session information. For instance, services typically restrict access to information, and therefore need to authenticate users. Authentication should be done once per session, and further interactions in the session should not require reauthentication.

To create the view of such sessions, extra information has to be stored at the client, and returned with each request in a session, for a server to identify that a request is part of a user session. Extra information about the session also has to be maintained at the server.

This extra information is maintained in the form of a **cookie** at the client; a cookie is simply a small piece of text containing identifying information. The server sends a cookie to the client after authentication, and also keeps a copy locally. Cookies sent to different clients contain different identifying text. The browser sends the cookie automatically on further document requests from the same server. By comparing the cookie with locally stored cookies at the server, the server can identify the request as

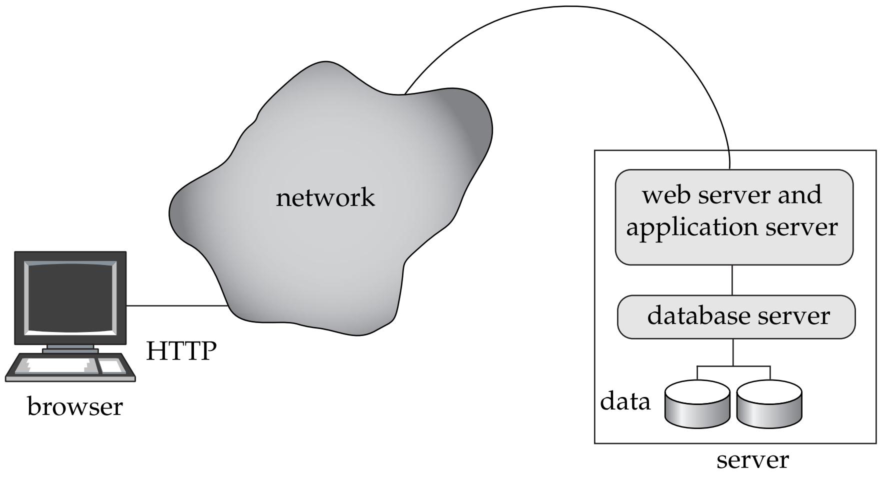


Figure 21.4 Two-tier Web architecture.

part of an ongoing session. Cookies can also be used for storing user preferences and using them when the server replies to a request. Cookies can be stored permanently at the browser; they identify the user on subsequent visits to the same site, without any identification information being typed in.

21.1.4 Servlets

In a two-tier Web architecture, the application runs as part of the Web server itself. One way of implementing such an architecture is to load Java programs with the Web server. The Java **servlet** specification defines an application programming interface for communication between the Web server and the application program. The word *servlet* also refers to a Java program that implements the servlet interface. The program is loaded into the Web server when the server starts up or when the server receives a Web request for executing the servlet application. Figure 21.5 is an example of servlet code to implement the form in Figure 21.1.

The servlet is called BankQueryServlet, while the form specifies that action="BankQuery". The Web server must be told that this servlet is to be used to handle requests for BankQuery.

The example will give you an idea of how servlets are used. For details needed to build your own servlet application, you can consult a book on servlets or read the online documentation on servlets that is part of the Java documentation from Sun. See the bibliographical notes for references to these sources.

The form specifies that the HTTP get mechanism is used for transmitting parameters (post is the other widely used mechanism). So the doGet() method of the servlet, which is defined in the code, gets invoked. Each request results in a new thread within which the call is executed, so multiple requests can be handled in parallel.

Any values from the form menus and input fields on the Web page, as well as cookies, pass through an object of the HttpServletRequest class that is created for the

```
public class BankQueryServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse result)
        throws ServletException, IOException
    {
        String type = request.getParameter("type");
        String number = request.getParameter("number");
        ... code to find the loan amount/account balance ...
        ... using JDBC to communicate with the database ..
        ... we assume the value is stored in the variable balance

        result.setContentType("text/html");
        PrintWriter out = result.getWriter();
        out.println("<HEAD><TITLE> Query Result</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("Balance on " + type + number + " = " + balance);
        out.println("</BODY>");
        out.close();
    }
}
```

Figure 21.5 Example of servlet code.

request, and the reply to the request passes through an object of the class HttpServlet-Response.[1]

The doGet() method code in the example extracts values of the parameter's type and number by using request.getParameter(), and uses these to run a query against a database. The code used to access the database is not shown; refer to Section 4.13.2 for details of how to use JDBC to access a database. The system returns the results of the query to the requester by printing them out in HTML format to the HttpServlet-Response result.

The servlet API provides a convenient method of creating sessions. Invoking the method getSession(true) of the class HttpServletRequest creates a new object of type HttpSession if this is the first request from that client; the argument true says that a session must be created if the request is a new request. The method returns an existing object if it had been created already for that browser session. Internally, cookies are used to recognize that a request is from the same browser session as an earlier request. The servlet code can store and look up (attribute-name, value) pairs in the HttpSession object, to maintain state across multiple requests. For instance, the first request in a session may ask for a user-id and password, and store the user-id in the session object. On subsequent requests from the browser session, the user-id will be found in the session object.

Displaying a set of results from a query is a common task for many database applications. It is possible to build a generic function that will take any JDBC ResulSet as argument, and display the tuples in the ResulSet appropriately. JDBC metadata calls

1. The servlet interface can also support non-HTTP requests, although our examples only use HTTP.

can be used to find information such as the number of columns, and the name and types of the columns, in the query result; this information is then used to print the query result.

21.1.5 Server-Side Scripting

Writing even a simple Web application in a programming language such as Java or C is a rather time-consuming task that requires many lines of code and programmers familiar with the intricacies of the language. An alternative approach, that of **server-side scripting**, provides a much easier method for creating many applications. Scripting languages provide constructs that can be embedded within HTML documents. In server-side scripting, before delivering a Web page, the server executes the scripts embedded within the HTML contents of the page. Each piece of script, when executed, can generate text that is added to the page (or may even delete content from the page). The source code of the scripts is removed from the page, so the client may not even be aware that the page orignally had any code in it. The executed script may contain SQL code that is executed against a database.

Several scripting languages have appeared in recent years. These include Server-Side Javascript from Netscape, JScript from Microsoft, JavaServer Pages (JSP) from Sun, the PHP Hypertext Preprocessor (PHP), ColdFusion's ColdFusion Markup Language (CFML) and Zope's DTML. In fact, it is even possible to embed code written in older scripting languages such as VBScript, Perl, and Python into HTML pages. For instance, Microsoft's Active Server Pages (ASP) supports embedded VBScript and JScript. Other approaches have extended report-writer software, originally developed for generating printable reports, to generate HTML reports. These also support HTML forms for getting parameter values that are used in the queries embedded in the reports.

Clearly, there are many options from which to choose. They all support similar features, but differ in the style of programming and the ease with which simple applications can be created.

21.1.6 Improving Performance

Web sites may be accessed by millions or billions of people from across the globe, at rates of thousands of requests per second, or even more, for the most popular sites. Ensuring that requests are served with low response times is a major challenge for Web site developers.

Caching techniques of various types are used to exploit commonalities between transactions. For instance, suppose the application code for servicing each request needs to contact a database through JDBC. Creating a new JDBC connection may take several milliseconds, so opening a new connection for each request is not a good idea if very high transaction rates are to be supported. Many applications create a pool of open JDBC connections, and each request uses one of the connections from the pool.

Many requests may result in exactly the same query being executed on the database. The cost of communication with the database can be greatly reduced by caching the results of earlier queries, and reusing them, so long as the query result has not changed at the database. Some Web servers support such query result caching.

Costs can be further reduced by caching the final Web page that is sent in response to a request. If a new request comes with exactly the same parameters as a previous request, if the resultant Web page is in the cache it can be reused, avoiding the cost of recomputing the page.

Cached query results and cached Web pages are forms of materialized views. If the underlying database data changes, they can be discarded, or can be recomputed, or even incrementally updated, as in materialized view maintenance (Section 14.5). For example, the IBM Web server that was used in the 2000 Olympics can keep track of what data a cached Web page depends on and recompute the page if the data change.

21.2 Performance Tuning

Tuning the performance of a system involves adjusting various parameters and design choices to improve its performance for a specific application. Various aspects of a database-system design—ranging from high-level aspects such as the schema and transaction design, to database parameters such as buffer sizes, down to hardware issues such as number of disks—affect the performance of an application. Each of these aspects can be adjusted so that performance is improved.

21.2.1 Location of Bottlenecks

The performance of most systems (at least before they are tuned) is usually limited primarily by the performance of one or a few components, called **bottlenecks**. For instance, a program may spend 80 percent of its time in a small loop deep in the code, and the remaining 20 percent of the time on the rest of the code; the small loop then is a bottleneck. Improving the performance of a component that is not a bottleneck does little to improve the overall speed of the system; in the example, improving the speed of the rest of the code cannot lead to more than a 20 percent improvement overall, whereas improving the speed of the bottleneck loop could result in an improvement of nearly 80 percent overall, in the best case.

Hence, when tuning a system, we must first try to discover what are the bottlenecks, and then to eliminate the bottlenecks by improving the performance of the components causing them. When one bottleneck is removed, it may turn out that another component becomes the bottleneck. In a well-balanced system, no single component is the bottleneck. If the system contains bottlenecks, components that are not part of the bottleneck are underutilized, and could perhaps have been replaced by cheaper components with lower performance.

For simple programs, the time spent in each region of the code determines the overall execution time. However, database systems are much more complex, and can be modeled as **queueing systems**. A transaction requests various services from the database system, starting from entry into a server process, disk reads during execution, CPU cycles, and locks for concurrency control. Each of these services has a queue associated with it, and small transactions may spend most of their time waiting in queues—especially in disk I/O queues—instead of executing code. Figure 21.6 illustrates some of the queues in a database system.

As a result of the numerous queues in the database, bottlenecks in a database system typically show up in the form of long queues for a particular service, or, equivalently, in high utilizations for a particular service. If requests are spaced exactly uniformly, and the time to service a request is less than or equal to the time before the next request arrives, then each request will find the resource idle and can therefore start execution immediately without waiting. Unfortunately, the arrival of requests in a database system is never so uniform, and is instead random.

If a resource, such as a disk, has a low utilization, then, when a request is made, the resource is likely to be idle, in which case the waiting time for the request will be 0. Assuming uniformly randomly distributed arrivals, the length of the queue (and correspondingly the waiting time) go up exponentially with utilization; as utilization approaches 100 percent, the queue length increases sharply, resulting in excessively long waiting times. The utilization of a resource should be kept low enough that queue length is short. As a rule of the thumb, utilizations of around 70 percent are considered to be good, and utilizations above 90 percent are considered excessive, since they will result in significant delays. To learn more about the theory of queueing systems, generally referred to as **queueing theory**, you can consult the references cited in the bibliographical notes.

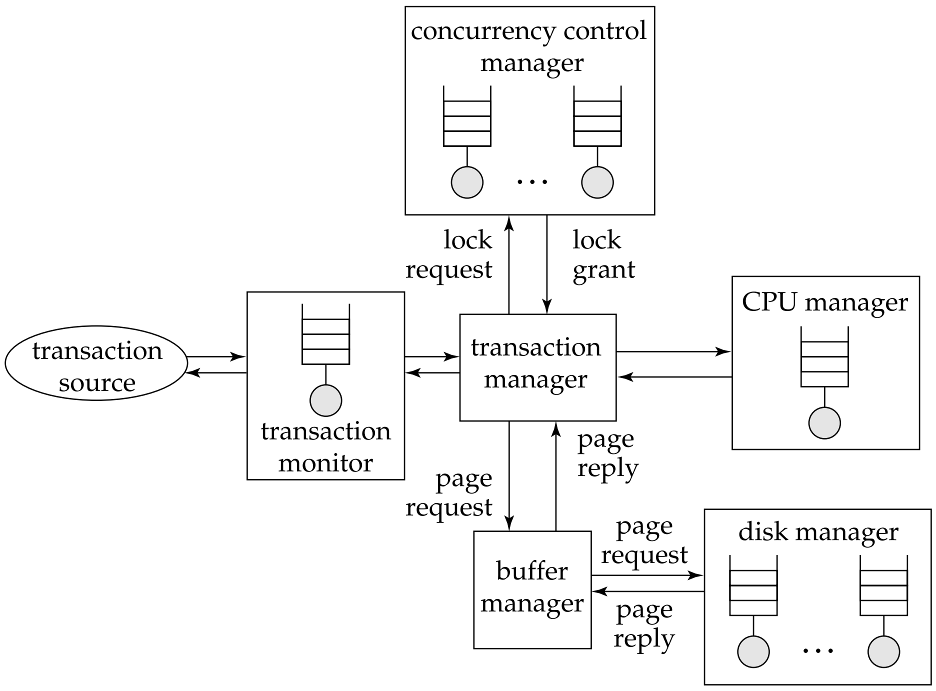


Figure 21.6 Queues in a database system.

21.2.2 Tunable Parameters

Database administrators can tune a database system at three levels. The lowest level is at the hardware level. Options for tuning systems at this level include adding disks or using a RAID system if disk I/O is a bottleneck, adding more memory if the disk buffer size is a bottleneck, or moving to a faster processor if CPU use is a bottleneck.

The second level consists of the database-system parameters, such as buffer size and checkpointing intervals. The exact set of database-system parameters that can be tuned depends on the specific database system. Most database-system manuals provide information on what database-system parameters can be adjusted, and how you should choose values for the parameters. Well-designed database systems perform as much tuning as possible automatically, freeing the user or database administrator from the burden. For instance, in many database systems the buffer size is fixed but tunable. If the system automatically adjusts the buffer size by observing indicators such as page-fault rates, then the user will not have to worry about tuning the buffer size.

The third level is the highest level. It includes the schema and transactions. The administrator can tune the design of the schema, the indices that are created, and the transactions that are executed, to improve performance. Tuning at this level is comparatively system independent.

The three levels of tuning interact with one another; we must consider them together when tuning a system. For example, tuning at a higher level may result in the hardware bottleneck changing from the disk system to the CPU, or vice versa.

21.2.3 Tuning of Hardware

Even in a well-designed transaction processing system, each transaction usually has to do at least a few I/O operations, if the data required by the transaction is on disk. An important factor in tuning a transaction processing system is to make sure that the disk subsystem can handle the rate at which I/O operations are required. For instance, disks today have an access time of about 10 milliseconds, and transfer times of 20 MB per second, which gives about 100 random access I/O operations of 1 KB each. If each transaction requires just 2 I/O operations, a single disk would support at most 50 transactions per second. The only way to support more transactions per second is to increase the number of disks. If the system needs to support n transactions per second, each performing 2 I/O operations, data must be striped (or otherwise partitioned) across $n/50$ disks (ignoring skew).

Notice here that the limiting factor is not the capacity of the disk, but the speed at which random data can be accessed (limited in turn by the speed at which the disk arm can move). The number of I/O operations per transaction can be reduced by storing more data in memory. If all data are in memory, there will be no disk I/O except for writes. Keeping frequently used data in memory reduces the number of disk I/Os, and is worth the extra cost of memory. Keeping very infrequently used data in memory would be a waste, since memory is much more expensive than disk.

The question is, for a given amount of money available for spending on disks or memory, what is the best way to spend the money to achieve maximum number of transactions per second. A reduction of 1 I/O per second saves (price per disk drive) / (access per second per disk). Thus, if a particular page is accessed n times per second, the saving due to keeping it in memory is n times the above value. Storing a page in memory costs (price per MB of memory) / (pages per MB of memory). Thus, the break-even point is

$$n * \frac{\textit{price per disk drive}}{\textit{access per second per disk}} = \frac{\textit{price per MB of memory}}{\textit{pages per MB of memory}}$$

We can rearrange the equation, and substitute current values for each of the above parameters to get a value for n; if a page is accessed more frequently than this, it is worth buying enough memory to store it. Current disk technology and memory and disk prices give a value of n around $1/300$ times per second (or equivalently, once in 5 minutes) for pages that are randomly accessed.

This reasoning is captured by the rule of thumb called the **5-minute rule**: If a page is used more frequently than once in 5 minutes, it should be cached in memory. In other words, it is worth buying enough memory to cache all pages that are accessed at least once in 5 minutes on an average. For data that are accessed less frequently, buy enough disks to support the rate of I/O required for the data.

The formula for finding the break-even point depends on factors, such as the costs of disks and memory, that have changed by factors of 100 or 1000 over the past decade. However, it is interesting to note that the ratios of the changes have been such that the break-even point has remained at roughly 5 minutes; the 5-minute rule has not changed to say, a 1-hour rule or a 1-second rule!

For data that are sequentially accessed, significantly more pages can be read per second. Assuming 1 MB of data is read at a time, we get the **1-minute rule**, which says that sequentially accessed data should be cached in memory if they are used at least once in 1 minute.

The rules of thumb take only number of I/O operations into account, and do not consider factors such as response time. Some applications need to keep even infrequently used data in memory, to support response times that are less than or comparable to disk access time.

Another aspect of tuning is in whether to use RAID 1 or RAID 5. The answer depends on how frequently the data are updated, since RAID 5 is much slower than RAID 1 on random writes: RAID 5 requires 2 reads and 2 writes to execute a single random write request. If an application performs r random reads and w random writes per second to support a particular throughput, a RAID 5 implementation would require $r + 4w$ I/O operations per second whereas a RAID 1 implementation would require $r + w$ I/O operations per second. We can then calculate the number of disks required to support the required I/O operations per second by dividing the result of the calculation by 100 I/O operations per second (for current generation disks). For many applications, r and w are large enough that the $(r + w)/100$ disks can easily hold two copies of all the data. For such applications, if RAID 1 is used, the required number of disks is actually less than the required number of disks if RAID 5 is used! Thus RAID 5 is useful only when the data storage requirements are very large, but the I/O rates and data transfer requirements are small, that is, for very large and very "cold" data.

21.2.4 Tuning of the Schema

Within the constraints of the chosen normal form, it is possible to partition relations vertically. For example, consider the *account* relation, with the schema

account (*account-number, branch-name, balance*)

for which *account-number* is a key. Within the constraints of the normal forms (BCNF and third normal forms), we can partition the *account* relation into two relations:

account-branch (*account-number, branch-name*)
account-balance (*account-number, balance*)

The two representations are logically equivalent, since *account-number* is a key, but they have different performance characteristics.

If most accesses to account information look at only the *account-number* and *balance*, then they can be run against the *account-balance* relation, and access is likely to be somewhat faster, since the *branch-name* attribute is not fetched. For the same reason, more tuples of *account-balance* will fit in the buffer than corresponding tuples of *account*, again leading to faster performance. This effect would be particularly marked if the *branch-name* attribute were large. Hence, a schema consisting of *account-branch* and *account-balance* would be preferable to a schema consisting of the *account* relation in this case.

On the other hand, if most accesses to account information require both *balance* and *branch-name*, using the *account* relation would be preferable, since the cost of the join of *account-balance* and *account-branch* would be avoided. Also, the storage overhead would be lower, since there would be only one relation, and the attribute *account-number* would not be replicated.

Another trick to improve performance is to store a **denormalized relation**, such as a join of *account* and *depositor*, where the information about branch-names and balances is repeated for every account holder. More effort has to be expended to make sure the relation is consistent whenever an update is carried out. However, a query that fetches the names of the customers and the associated balances will be speeded up, since the join of *account* and *depositor* will have been precomputed. If such a query is executed frequently, and has to be performed as efficiently as possible, the denormalized relation could be beneficial.

Materialized views can provide the benefits that denormalized relations provide, at the cost of some extra storage; we describe performance tuning of materialized views in Section 21.2.6. A major advantage to materialized views over denormalized relations is that maintaining consistency of redundant data becomes the job of the database system, not the programmer. Thus, materialized views are preferable, whenever they are supported by the database system.

Another approach to speed up the computation of the join without materializing it, is to cluster records that would match in the join on the same disk page. We saw such clustered file organizations in Section 11.7.2.

21.2.5 Tuning of Indices

We can tune the indices in a system to improve performance. If queries are the bottleneck, we can often speed them up by creating appropriate indices on relations. If updates are the bottleneck, there may be too many indices, which have to be updated when the relations are updated. Removing indices may speed up certain updates.

The choice of the type of index also is important. Some database systems support different kinds of indices, such as hash indices and B-tree indices. If range queries are common, B-tree indices are preferable to hash indices. Whether to make an index a clustered index is another tunable parameter. Only one index on a relation can be made clustered, by storing the relation sorted on the index attributes. Generally, the index that benefits the most number of queries and updates should be made clustered.

To help identify what indices to create, and which index (if any) on each relation should be clustered, some database systems provide *tuning wizards*. These tools use the past history of queries and updates (called the *workload*) to estimate the effects of various indices on the execution time of the queries and updates in the workload. Recommendations on what indices to create are based on these estimates.

21.2.6 Using Materialized Views

Maintaining materialized views can greatly speed up certain types of queries, in particular aggregate queries. Recall the example from Section 14.5 where the total loan amount at each branch (obtained by summing the loan amounts of all loans at the branch) is required frequently. As we saw in that section, creating a materialized view storing the total loan amount for each branch can greatly speed up such queries.

Materialized views should be used with care, however, since there is not only a space overhead for storing them but, more important, there is also a time overhead for maintaining materialized views. In the case of **immediate view maintenance**, if the updates of a transaction affect the materialized view, the materialized view must be updated as part of the same transaction. The transaction may therefore run slower. In the case of **deferred view maintenance**, the materialized view is updated later; until it is updated, the materialized view may be inconsistent with the database relations. For instance, the materialized view may be brought up-to-date when a query uses the view, or periodically. Using deferred maintenance reduces the burden on update transactions.

An important question is, how does one select which materialized views to maintain? The system administrator can make the selection manually by examining the types of queries in the workload, and finding out which queries need to run faster and which updates/queries may be executed slower. From the examination, the system administrator may choose an appropriate set of materialized views. For instance, the administrator may find that a certain aggregate is used frequently, and choose to materialize it, or may find that a particular join is computed frequently, and choose to materialize it.

However, manual choice is tedious for even moderately large sets of query types, and making a good choice may be difficult, since it requires understanding the costs of different alternatives; only the query optimizer can estimate the costs with reasonable accuracy, without actually executing the query. Thus a good set of views may only be found by trial and error—that is, by materializing one or more views, running the workload, and measuring the time taken to run the queries in the workload. The administrator repeats the process until a set of views is found that gives acceptable performance.

A better alternative is to provide support for selecting materialized views within the database system itself, integrated with the query optimizer. Some database systems, such as Microsoft SQL Server 7.5 and the RedBrick Data Warehouse from Informix, provide tools to help the database administrator with index and materialized view selection. These tools examine the workload (the history of queries and updates) and suggest indices and views to be materialized. The user may specify the importance of speeding up different queries, which the administrator takes into account when selecting views to materialize.

Microsoft's materialized view selection tool also permits the user to ask "what if" questions, whereby the user can pick a view, and the optimizer then estimates the effect of materializing the view on the total cost of the workload and on the individual costs of different query/update types in the workload.

In fact, even automated selection techniques are implemented in a similar manner internally: Different alternatives are tried, and for each the query optimizer estimates the costs and benefits of materializing it.

Greedy heuristics for materialized view selection operate roughly this way: They estimate the benefits of materializing different views, and choose the view that gives either the maximum benefit or the maximum benefit per unit space (that is, benefit divided by the space required to store the view). Once the heuristic has selected a view, the benefits of other views may have changed, so the heuristic recomputes these, and chooses the next best view for materialization. The process continues until either the available disk space for storing materialized views is exhausted, or the cost of view maintenance increases above acceptable limits.

21.2.7 Tuning of Transactions

In this section, we study two approaches for improving transaction performance:

- Improve set orientation
- Reduce lock contention

In the past, optimizers on many database systems were not particularly good, so how a query was written would have a big influence on how it was executed, and therefore on the performance. Today's advanced optimizers can transform even badly written queries and execute them efficiently, so the need for tuning individual queries is less important than it used to be. However, complex queries containing nested subqueries are not optimized very well by many optimizers. Most systems provide a mechanism to find out the exact execution plan for a query; this information can be used to rewrite the query in a form that the optimizer can deal with better.

In embedded SQL, if a query is executed frequently with different values for a parameter, it may help to combine the calls into a more set-oriented query that is executed only once. The costs of communication of SQL queries can be high in client−server systems, so combining the embedded SQL calls is particularly helpful in such systems.

For example, consider a program that steps through each department specified in a list, invoking an embedded SQL query to find the total expenses of the department by using the **group by** construct on a relation *expenses*(*date*, *employee*, *department*, *amount*). If the *expenses* relation does not have a clustered index on *department*, each such query will result in a scan of the relation. Instead, we can use a single SQL query to find total expenses of all departments; the query can be evaluated with a single scan. The relevant departments can then be looked up in this (much smaller) temporary relation containing the aggregate. Even if there is an index that permits efficient access to tuples of a given department, using multiple SQL queries can have a high communication overhead in a client−server system. Communication cost can be reduced by using a single SQL query, fetching its results to the client side, and then stepping through the results to find the required tuples.

Another technique used widely in client−server systems to reduce the cost of communication and SQL compilation is to use stored procedures, where queries are stored at the server in the form of procedures, which may be precompiled. Clients can invoke these stored procedures, rather than communicate entire queries.

Concurrent execution of different types of transactions can sometimes lead to poor performance because of contention on locks. Consider, for example, a banking database. During the day, numerous small update transactions are executed almost continuously. Suppose that a large query that computes statistics on branches is run at the same time. If the query performs a scan on a relation, it may block out all updates on the relation while it runs, and that can have a disastrous effect on the performance of the system.

Some database systems—Oracle, for example—permit multiversion concurrency control, whereby queries are executed on a snapshot of the data, and updates can go on concurrently. This feature should be used if available. If it is not available, an alternative option is to execute large queries at times when updates are few or nonexistent. For databases supporting Web sites, there may be no such quiet period for updates.

Another alternative is to use weaker levels of consistency, whereby evaluation of the query has a minimal impact on concurrent updates, but the query results are not guaranteed to be consistent. The application semantics determine whether approximate (inconsistent) answers are acceptable.

Long update transactions can cause performance problems with system logs, and can increase the time taken to recover from system crashes. If a transaction performs many updates, the system log may become full even before the transaction completes, in which case the transaction will have to be rolled back. If an update transaction runs for a long time (even with few updates), it may block deletion of old parts of the log, if the logging system is not well designed. Again, this blocking could lead to the log getting filled up.

To avoid such problems, many database systems impose strict limits on the number of updates that a single transaction can carry out. Even if the system does not impose such limits, it is often helpful to break up a large update transaction into a set of smaller update transactions where possible. For example, a transaction that gives a raise to every employee in a large corporation could be split up into a series of small transactions, each of which updates a small range of employee-ids. Such transactions are called **minibatch transactions**. However, minibatch transactions must be used with care. First, if there are concurrent updates on the set of employees, the result of the set of smaller transactions may not be equivalent to that of the single large transaction. Second, if there is a failure, the salaries of some of the employees would have been increased by committed transactions, but salaries of other employees would not. To avoid this problem, as soon as the system recovers from failure, we must execute the transactions remaining in the batch.

21.2.8 Performance Simulation

To test the performance of a database system even before it is installed, we can create a performance-simulation model of the database system. Each service shown in Figure 21.6, such as the CPU, each disk, the buffer, and the concurrency control, is modeled in the simulation. Instead of modeling details of a service, the simulation model may capture only some aspects of each service, such as the **service time**—that is, the time taken to finish processing a request once processing has begun. Thus, the simulation can model a disk access from just the average disk access time.

Since requests for a service generally have to wait their turn, each service has an associated queue in the simulation model. A transaction consists of a series of requests. The requests are queued up as they arrive, and are serviced according to the policy for that service, such as first come, first served. The models for services such as CPU and the disks conceptually operate in parallel, to account for the fact that these subsystems operate in parallel in a real system.

Once the simulation model for transaction processing is built, the system administrator can run a number of experiments on it. The administrator can use experiments with simulated transactions arriving at different rates to find how the system would behave under various load conditions. The administrator could run other experiments that vary the service times for each service to find out how sensitive the performance is to each of them. System parameters, too, can be varied, so that performance tuning can be done on the simulation model.

21.3 Performance Benchmarks

As database servers become more standardized, the differentiating factor among the products of different vendors is those products' performance. **Performance benchmarks** are suites of tasks that are used to quantify the performance of software systems.

21.3.1 Suites of Tasks

Since most software systems, such as databases, are complex, there is a good deal of variation in their implementation by different vendors. As a result, there is a significant amount of variation in their performance on different tasks. One system may be the most efficient on a particular task; another may be the most efficient on a different task. Hence, a single task is usually insufficient to quantify the performance of the system. Instead, the performance of a system is measured by suites of standardized tasks, called *performance benchmarks*.

Combining the performance numbers from multiple tasks must be done with care. Suppose that we have two tasks, T_1 and T_2, and that we measure the throughput of a system as the number of transactions of each type that run in a given amount of time —say, 1 second. Suppose that system A runs T_1 at 99 transactions per second, and that T_2 runs at 1 transaction per second. Similarly, let system B run both T_1 and T_2 at 50 transactions per second. Suppose also that a workload has an equal mixture of the two types of transactions.

If we took the average of the two pairs of numbers (that is, 99 and 1, versus 50 and 50), it might appear that the two systems have equal performance. However, it is *wrong* to take the averages in this fashion—if we ran 50 transactions of each type, system *A* would take about 50.5 seconds to finish, whereas system *B* would finish in just 2 seconds!

The example shows that a simple measure of performance is misleading if there is more than one type of transaction. The right way to average out the numbers is to take the **time to completion** for the workload, rather than the average **throughput** for each transaction type. We can then compute system performance accurately in transactions per second for a specified workload. Thus, system A takes $50.5/100$, which is 0.505 seconds per transaction, whereas system B takes 0.02 seconds per transaction, on average. In terms of throughput, system A runs at an average of 1.98 transactions per second, whereas system B runs at 50 transactions per second. Assuming that transactions of all the types are equally likely, the correct way to average out the throughputs on different transaction types is to take the **harmonic mean** of the throughputs. The harmonic mean of n throughputs $t_1, \ldots, t_n$ is defined as

$$\frac{n}{\frac{1}{t_1} + \frac{1}{t_2} + \cdots + \frac{1}{t_n}}$$

For our example, the harmonic mean for the throughputs in system A is 1.98. For system B, it is 50. Thus, system B is approximately 25 times faster than system A on a workload consisting of an equal mixture of the two example types of transactions.

21.3.2 Database-Application Classes

Online transaction processing (**OLTP**) and **decision support** (including **online analytical processing** (**OLAP**)) are two broad classes of applications handled by database systems. These two classes of tasks have different requirements. High concurrency and clever techniques to speed up commit processing are required for supporting a high rate of update transactions. On the other hand, good query-evaluation algorithms and query optimization are required for decision support. The architecture of some database systems has been tuned to transaction processing; that of others, such as the Teradata DBC series of parallel database systems, has been tuned to decision support. Other vendors try to strike a balance between the two tasks.

Applications usually have a mixture of transaction-processing and decision- support requirements. Hence, which database system is best for an application depends on what mix of the two requirements the application has.

Suppose that we have throughput numbers for the two classes of applications separately, and the application at hand has a mix of transactions in the two classes. We must be careful even about taking the harmonic mean of the throughput numbers, because of **interference** between the transactions. For example, a long-running decision-support transaction may acquire a number of locks, which may prevent all progress of update transactions. The harmonic mean of throughputs should be used only if the transactions do not interfere with one another.

21.3.3 The TPC Benchmarks

The **Transaction Processing Performance Council** (**TPC**), has defined a series of benchmark standards for database systems.

The TPC benchmarks are defined in great detail. They define the set of relations and the sizes of the tuples. They define the number of tuples in the relations not as a fixed number, but rather as a multiple of the number of claimed transactions per second, to reflect that a larger rate of transaction execution is likely to be correlated with a larger number of accounts. The performance metric is throughput, expressed as **transactions per second** (**TPS)**. When its performance is measured, the system must provide a response time within certain bounds, so that a high throughput cannot be obtained at the cost of very long response times. Further, for business applications, cost is of great importance. Hence, the TPC benchmark also measures performance in terms of **price per TPS**. A large system may have a high number of transactions per second, but may be expensive (that is, have a high price per TPS). Moreover, a company cannot claim TPC benchmark numbers for its systems *without* an external audit that ensures that the system faithfully follows the definition of the benchmark, including full support for the ACID properties of transactions.

The first in the series was the **TPC-A benchmark**, which was defined in 1989. This benchmark simulates a typical bank application by a single type of transaction that models cash withdrawal and deposit at a bank teller. The transaction updates several relations—such as the bank balance, the teller's balance, and the customer's balance—and adds a record to an audit trail relation. The benchmark also incorporates communication with terminals, to model the end-to-end performance of the system realistically. The **TPC-B benchmark** was designed to test the core performance of the database system, along with the operating system on which the system runs. It removes the parts of the TPC-A benchmark that deal with users, communication, and terminals, to focus on the back-end database server. Neither TPC-A nor TPC-B is widely used today.

The **TPC-C benchmark** was designed to model a more complex system than the TPC-A benchmark. The TPC-C benchmark concentrates on the main activities in an order-entry environment, such as entering and delivering orders, recording payments, checking status of orders, and monitoring levels of stock. The TPC-C benchmark is still widely used for transaction processing.

The **TPC-D benchmark** was designed to test the performance of database systems on decision-support queries. Decision-support systems are becoming increasingly important today. The TPC-A, TPC-B, and TPC-C benchmarks measure performance on transaction-processing workloads, and should not be used as a measure of performance on decision-support queries. The D in TPC-D stands for **decision support**. The TPC-D benchmark schema models a sales/distribution application, with parts, suppliers, customers, and orders, along with some auxiliary information. The sizes of the relations are defined as a ratio, and database size is the total size of all the relations, expressed in gigabytes. TPC-D at scale factor 1 represents the TPC-D benchmark on a 1-gigabyte database, while scale factor 10 represents a 10-gigabyte database. The benchmark workload consists of a set of 17 SQL queries modeling common tasks executed on decision-support systems. Some of the queries make use of complex SQL features, such as aggregation and nested queries.

The benchmark's users soon realized that the various TPC-D queries could be significantly speeded up by using materialized views and other redundant information. There are applications, such as periodic reporting tasks, where the queries are known in advance and materialized view can be carefully selected to speed up the queries. It is necessary, however, to account for the overhead of maintaining materalized views.

The **TPC-R benchmark** (where R stands for **reporting**) is a refinement of the TPC-D benchmark. The schema is the same, but there are 22 queries, of which 16 are from TPC-D. In addition, there are two updates, a set of inserts and a set of deletes. The database running the benchmark is permitted to use materialized views and other redundant information.

In contrast the **TPC-H benchmark** (where H represents **ad hoc**) uses the same schema and workload as TPC-R but prohibits materialized views and other redundant information, and permits indices only on primary and foreign keys. This benchmark models ad hoc querying where the queries are not known beforehand, so it is not possible to create appropriate materialized views ahead of time.

Both TPC-H and TPC-R measure performance in this way: The **power test** runs the queries and updates one at a time sequentially, and 3600 seconds divided by geometric mean of the execution times of the queries (in seconds) gives a measure of queries per hour. The **throughput test** runs multiple streams in parallel, with each stream executing all 22 queries. There is also a parallel update stream. Here the total time for the entire run is used to compute the number of queries per hour.

The **composite query per hour metric**, which is the overall metric, is then obtained as the square root of the the product of the power and throughput metrics. A **composite price/performance metric** is defined by dividing the system price by the composite metric.

The **TPC-W** Web commerce benchmark is an end-to-end benchmark that models Web sites having static content (primarily images) and dynamic content generated from a database. Caching of dynamic content is specifically permitted, since it is very useful for speeding up Web sites. The benchmark models an electronic bookstore, and like other TPC benchmarks, provides for different scale factors. The primary performance metrics are **Web interactions per second (WIPS)** and price per WIPS.

21.3.4 The OODB Benchmarks

The nature of applications in an object-oriented database, OODB, is different from that of typical transaction-processing applications. Therefore, a different set of benchmarks has been proposed for OODBs. The Object Operations benchmark, version 1, popularly known as the **OO1 benchmark**, was an early proposal. The **OO7 benchmark** follows a philosophy different from that of the TPC benchmarks. The TPC benchmarks provide one or two numbers (in terms of average transactions per second, and transactions per second per dollar); the OO7 benchmark provides a set of numbers, containing a separate benchmark number for each of several different kinds of operations. The reason for this approach is that it is not yet clear what is the *typical* OODB transaction. It is clear that such a transaction will carry out certain operations, such as traversing a set of connected objects or retrieving all objects in a class, but it is not clear exactly what mix of these operations will be used. Hence, the benchmark provides separate numbers for each class of operations; the numbers can be combined in an appropriate way, depending on the specific application.

21.4 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 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 used to establish 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 give 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.

21.4.1 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 latest version of the SQL standard, called SQL:1999, adds a variety of features to SQL. We have seen many of these features in earlier chapters, and will see a few in later chapters. The standard is broken into several parts:

- SQL/Framework (Part 1) provides an overview of the standard.
- SQL/Foundation (Part 2) 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.
- SQL/CLI (Call Level Interface) (Part 3) defines application program interfaces to SQL.
- SQL/PSM (Persistent Stored Modules) (Part 4) defines extensions to SQL to make it procedural.
- SQL/Bindings (Part 5) defines standards for embedded SQL for different embedding languages.

The SQL:1999 OLAP features (Section 22.2.3) have been specified as an amendment to the earlier version of the SQL:1999 standard. There are several other parts under development, including

- Part 7: SQL/Temporal deals with standards for temporal data.
- Part 9: SQL/MED (Management of External Data) defines standards for interfacing an SQL system to external sources. By writing wrappers, system 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.

The missing numbers (Parts 6 and 8) cover features such as distributed transaction processing and multimedia data, for which there is as yet no agreement on the standards. The multimedia standards propose to cover storage and retrieval of text data, spatial data, and still images.

21.4.2 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 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 transaction 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 differes 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 would 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.

21.4.3 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) is a group formed by OODB vendors to standardize the data model and language interfaces to OODBs. The C++ language interface specified by ODMG was discussed in Chapter 8. The ODMG has also specified a Java interface and a Smalltalk interface.

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

21.4.4 XML-Based Standards

A wide variety of standards based on XML (see Chapter 10) 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 defacto standards. RosettaNet, which falls into the former category, uses XML-based standards to facilitate supply-chain management in the computer and information technology industries. Companies such as Commerce One provide Web-based procurement systems, supply-chain management, and electonic marketplaces (including online auctions). BizTalk is a framework of XML schemas and guidelines, backed by Microsoft. These and other frameworks define catalogs, service descriptions, invoices, purchase orders, order status requests, shipping bills, and related items.

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 heterogeneous 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 is gaining wide acceptance in industry (including IBM and Microsoft). 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 other sites through SOAP. Microsoft has defined standards for accessing OLAP and mining data with SOAP. (OLAP and data mining are covered in Chapter 22.)

The W3C standard query language for XML is called *XQuery*. As of early 2001 the standard was in working draft stage, and should be finalized by the end of the year. Earlier XML query languages include *Quilt* (on which XQuery is based), *XML-QL*, and *XQL*.

21.5 E-Commerce∗∗

E-commerce refers to the process of carrying out various activities related to commerce, through electronic means, primarily through the internet. The types of activities include:

- Presale activities, needed to inform the potential buyer about the product or service being sold.
- The sale process, which includes negotiations on price and quality of service, and other contractual matters.
- The marketplace: When there are multiple sellers and buyers for a product, a marketplace, such as a stock exchange, helps in negotiating the price to be paid for the product. Auctions are used when there is a single seller and multiple buyers, and reverse auctions are used when there is a single buyer and multiple sellers.
- Payment for the sale.

- Activities related to delivery of the product or service. Some products and services can be delivered over the internet; for others the internet is used only for providing shipping information and for tracking shipments of products.
- Customer support and postsale service.

Databases are used extensively to support these activities. For some of the activities the use of databases is straightforward, but there are interesting application development issues for the other activities.

21.5.1 E-Catalogs

Any e-commerce site provides users with a catalog of the products and services that the site supplies. The services provided by an e-catalog may vary considerably.

At the minimum, an e-catalog must provide browsing and search facilities to help customers find the product they are looking for. To help with browsing, products should be organized into an intuitive hierarchy, so a few clicks on hyperlinks can lead a customer to the products they are interested in. Keywords provided by the customer (for example, "digital camera" or "computer") should speed up the process of finding required products. E-catalogs should also provide a means for customers to easily compare alternatives from which to choose among competing products.

E-catalogs can be customized for the customer. For instance, a retailer may have an agreement with a large company to supply some products at a discount. An employee of the company, viewing the catalog to purchase products for the company, should see prices as per the negotiated discount, instead of the regular prices. Because of legal restrictions on sales of some types of items, customers who are underage, or from certain states or countries, should not be shown items that cannot be legally sold to them. Catalogs can also be personalized to individual users, on the basis of past buying history. For instance, frequent customers may be offered special discounts on some items.

Supporting such customization requires customer information as well as special pricing/discount information and sales restriction information to be stored in a database. There are also challenges in supporting very high transaction rates, which are often tackled by caching of query results or generated Web pages.

21.5.2 Marketplaces

When there are multiple sellers or multiple buyers (or both) for a product, a marketplace helps in negotiating the price to be paid for the product. There are several different types of marketplaces:

- In a **reverse auction** system a buyer states requirements, and sellers bid for supplying the item. The supplier quoting the lowest price wins. In a closed bidding system, the bids are not made public, whereas in an open bidding system the bids are made public.
- In an **auction** there are multiple buyers and a single seller. For simplicity, assume that there is only one instance of each item being sold. Buyers bid for

the items being sold, and the highest bidder for an item gets to buy the item at the bid price.

When there are multiple copies of an item, things become more complicated: Suppose there are four items, and one bidder may want three copies for $10 each, while another wants two copies for $13 each. It is not possible to satisfy both bids. If the items will be of no value if they are not sold (for instance, airline seats, which must be sold before the plane leaves), the seller simply picks a set of bids that maximizes the income. Otherwise the decision is more complicated.

- In an **exchange**, such as a stock exchange, there are multiple sellers and multiple buyers. Buyers can specify the maximum price they are willing to pay, while sellers specify the minimum price they want. There is usually a *market maker* who matches buy and sell bids, deciding on the price for each trade (for instance, at the price of the sell bid).

There are other more complex types of marketplaces.

Among the database issues in handling marketplaces are these:

- Bidders need to be authenticated before they are allowed to bid.
- Bids (buy or sell) need to be recorded securely in a database. Bids need to be communicated quickly to other people involved in the marketplace (such as all the buyers or all the sellers), who may be numerous.
- Delays in broadcasting bids can lead to financial losses to some participants.
- The volumes of trades may be extremely large at times of stock market volatility, or toward the end of auctions. Thus, very high performance databases with large degrees of parallelism are used for such systems.

21.5.3 Order Settlement

After items have been selected (perhaps through an electronic catalog), and the price determined (perhaps by an electronic marketplace), the order has to be settled. Settlement involves payment for goods and the delivery of the goods.

A simple but unsecure way of paying electronically is to send a credit card number. There are two major problems. First, credit card fraud is possible. When a buyer pays for physical goods, companies can ensure that the address for delivery matches the card holder's address, so no one else can receive the goods, but for goods delivered electronically no such check is possible. Second, the seller has to be trusted to bill only for the agreed-on item and to not pass on the card number to unauthorized people who may misuse it.

Several protocols are available for secure payments that avoid both the problems listed above. In addition, they provide for better privacy, whereby the seller may not be given any unnecessary details about the buyer, and the credit card company is not provided any unnecessary information about the items purchased. All information transmitted must be encrypted so that anyone intercepting the data on the network cannot find out the contents. Public/private key encryption is widely used for this task.

The protocols must also prevent **person-in-the-middle attacks**, where someone can impersonate the bank or credit-card company, or even the seller, or buyer, and steal secret information. Impersonation can be perpetrated by passing off a fake key as someone else's public key (the bank's or credit-card company's, or the merchant's or the buyer's). Impersonation is prevented by a system of **digital certificates**, whereby public keys are signed by a certification agency, whose public key is well known (or which in turn has its public key certified by another certification agency and so on up to a key that is well known). From the well-known public key, the system can authenticate the other keys by checking the certificates in reverse sequence.

The **Secure Electronic Transaction** (**SET**) protocol is one such secure payment protocol. The protocol requires several rounds of communication between the buyer, seller, and the bank, in order to guarantee safety of the transaction.

There are also systems that provide for greater anonymity, similar to that provided by physical cash. The **DigiCash** payment system is one such system. When a payment is made in such a system, it is not possible to identify the purchaser. In contrast, identifying purchasers is very easy with credit cards, and even in the case of SET, it is possible to identify the purchaser with the cooperation of the credit card company or bank.

21.6 Legacy Systems

Legacy systems are older-generation systems that are incompatible with current-generation standards and systems. Such systems may still contain valuable data, and may support critical applications. The legacy systems of today are typically those built with technologies such as databases that use the network or hierarchical data models, or use Cobol and file systems without a database.

Porting legacy applications to a more modern environment is often costly in terms of both time and money, since they are often very large, consisting of millions of lines of code developed by teams of programmers, over several decades.

Thus, it is important to support these older-generation, or legacy, systems, and to facilitate their interoperation with newer systems. One approach used to interoperate between relational databases and legacy databases is to build a layer, called a **wrapper**, on top of the legacy systems that can make the legacy system appear to be a relational database. The wrapper may provide support for ODBC or other interconnection standards such as OLE-DB, which can be used to query and update the legacy system. The wrapper is responsible for converting relational queries and updates into queries and updates on the legacy system.

When an organization decides to replace a legacy system by a new system, it must follow a process called **reverse engineering**, which consists of going over the code of the legacy system to come up with schema designs in the required data model (such as an E-R model or an object-oriented data model). Reverse engineering also examines the code to find out what procedures and processes were implemented, in order to get a high-level model of the system. Reverse engineering is needed because legacy systems usually do not have high-level documentation of their schema and overall system design. When coming up with the design of a new system, the design is reviewed, so that it can be improved rather than just reimplemented as is. Extensive coding is required to support all the functionality (such as user interface and reporting systems) that were provided by the legacy system. The overall process is called **re-engineering**.

When a new system has been built and tested, the system must be populated with data from the legacy system, and all further activities must be carried out on the new system. However, abruptly transitioning to a new system, which is called the **big-bang approach**, carries several risks. First, users may not be familiar with the interfaces of the new system. Second there may be bugs or performance problems in the new system that were not discovered when it was tested. Such problems may lead to great losses for companies, since their ability to carry out critical transactions such as sales and purchases may be severely affected. In some extreme cases the new system has even been abandoned, and the legacy system reused, after an attempted switchover failed.

An alternative approach, called the **chicken-little approach**, incrementally replaces the functionality of the legacy system. For example, the new user interfaces may be used with the old system in the back end, or vice versa. Another option is to use the new system only for some functionality that can be decoupled from the legacy system. In either case, the legacy and new systems coexist for some time. There is therefore a need for developing and using wrappers on the legacy system to provide required functionality to interoperate with the new system. This approach, therefore has a higher development cost associated with it.

21.7 Summary

- The Web browser has emerged as the most widely used user interface to databases. HTML provides the ability to define interfaces that combine hyperlinks with forms facilities. Web browsers communicate with Web servers by the HTTP protocol. Web servers can pass on requests to application programs, and return the results to the browser.

- There are several client-side scripting languages—Javascript is the most widely used—that provide richer user interaction at the browser end.

- Web servers execute application programs to implement desired functionality. Servlets are a widely used mechanism to write application programs that run as part of the Web server process, in order to reduce overheads. There are also many server-side scripting languages that are interpreted by the Web server and provide application program functionality as part of the Web server.

- Tuning of the database-system parameters, as well as the higher-level database design—such as the schema, indices, and transactions—is important for good performance. Tuning is best done by identifying bottlenecks and eliminating them.

- Performance benchmarks play an important role in comparisons of database systems, especially as systems become more standards compliant. The TPC benchmark suites are widely used, and the different TPC benchmarks are useful for comparing the performance of databases under different workloads.
- Standards are important because of the complexity of database systems and their need for interoperation. Formal standards exist for SQL. Defacto standards, such as ODBC and JDBC, and standards adopted by industry groups, such as CORBA, have played an important role in the growth of client−server database systems. Standards for object-oriented databases, such as ODMG, are being developed by industry groups.
- E-commerce systems are fast becoming a core part of how commerce is performed. There are several database issues in e-commerce systems. Catalog management, especially personalization of the catalog, is done with databases. Electronic marketplaces help in pricing of products through auctions, reverse auctions, or exchanges. High-performance database systems are needed to handle such trading. Orders are settled by electronic payment systems, which also need high-performance database systems to handle very high transaction rates.
- Legacy systems are systems based on older-generation technologies such as nonrelational databases or even directly on file systems. Interfacing legacy systems with new-generation systems is often important when they run mission-critical systems. Migrating from legacy systems to new-generation systems must be done carefully to avoid disruptions, which can be very expensive.

Review Terms

- Web interfaces to databases
- HyperText Markup Language (HTML)
- Hyperlinks
- Uniform resource locator (URL)
- Client-side scripting
- Applets
- Client-side scripting language
- Web servers
- Session
- HyperText Transfer Protocol (HTTP)
- Common Gateway Interface (CGI)
- Connectionless
- Cookie
- Servlets
- Server-side scripting
- Performance tuning
- Bottlenecks
- Queueing systems
- Tunable parameters
- Tuning of hardware
- Five-minute rule
- One-minute rule
- Tuning of the schema
- Tuning of indices

- Materialized views
- Immediate view maintenance
- Deferred view maintenance
- Tuning of transactions
- Improving set orientedness
- Minibatch transactions
- Performance simulation
- Performance benchmarks
- Service time
- Time to completion
- Database-application classes
- The TPC benchmarks
  - ☐ TPC-A
  - ☐ TPC-B
  - ☐ TPC-C
  - ☐ TPC-D
  - ☐ TPC-R
  - ☐ TPC-H
  - ☐ TPC-W
- Web interactions per second
- OODB benchmarks
  - ☐ OO1
  - ☐ OO7
- Standardization
  - ☐ Formal standards
  - ☐ De facto standards
  - ☐ Anticipatory standards
  - ☐ Reactionary standards
- Database connectivity standards
  - ☐ ODBC
  - ☐ OLE-DB
  - ☐ X/Open XA standards
- Object database standards
  - ☐ ODMG
  - ☐ CORBA
- XML-based standards
- E-commerce
- E-catalogs
- Marketplaces
  - ☐ Auctions
  - ☐ Reverse-auctions
  - ☐ Exchange
- Order settlement
- Digital certificates
- Legacy systems
- Reverse engineering
- Re-engineering

Exercises

21.1 What is the main reason why servlets give better performance than programs that use the common gateway interface (CGI), even though Java programs generally run slower than C or C++ programs.

21.2 List some benefits and drawbacks of connectionless protocols over protocols that maintain connections.

21.3 List three ways in which caching can be used to speed up Web server performance.

21.4 **a.** What are the three broad levels at which a database system can be tuned to improve performance?
b. Give two examples of how tuning can be done, for each of the levels.

21.5 What is the motivation for splitting a long transaction into a series of small ones? What problems could arise as a result, and how can these problems be averted?

21.6 Suppose a system runs three types of transactions. Transactions of type A run at the rate of 50 per second, transactions of type B run at 100 per second, and transactions of type C run at 200 per second. Suppose the mix of transactions has 25 percent of type A, 25 percent of type B, and 50 percent of type C.

a. What is the average transaction throughput of the system, assuming there is no interference between the transactions.

b. What factors may result in interference between the transactions of different types, leading to the calculated throughput being incorrect?

21.7 Suppose the price of memory falls by half, and the speed of disk access (number of accesses per second) doubles, while all other factors remain the same. What would be the effect of this change on the 5 minute and 1 minute rule?

21.8 List some of the features of the TPC benchmarks that help make them realistic and dependable measures.

21.9 Why was the TPC-D benchmark replaced by the TPC-H and TPC-R benchmarks?

21.10 List some benefits and drawbacks of an anticipatory standard compared to a reactionary standard.

21.11 Suppose someone impersonates a company and gets a certificate from a certificate issuing authority. What is the effect on things (such as purchase orders or programs) certified by the impersonated company, and on things certified by other companies?

Project Suggestions

Each of the following is a large project, which can be a semester-long project done by a group of students. The difficulty of the project can be adjusted easily by adding or deleting features.

Project 21.1 Consider the E-R schema of Exercise 2.7 (Chapter 2), which represents information about teams in a league. Design and implement a Web-based system to enter, update, and view the data.

Project 21.2 Design and implement a shopping cart system that lets shoppers collect items into a shopping cart (you can decide what information is to be supplied for each item) and purchased together. You can extend and use the E-R schema of Exercise 2.12 of Chapter 2. You should check for availability of the item and deal with nonavailable items as you feel appropriate.

Project 21.3 Design and implement a Web-based system to record student registration and grade information for courses at a university.

Project 21.4 Design and implement a system that permits recording of course performance information—specifically, the marks given to each student in each assignment or exam of a course, and computation of a (weighted) sum of marks to get the total course marks. The number of assignments/exams should not be predefined; that is, more assignments/exams can be added at any time. The system should also support grading, permitting cutoffs to be specified for various grades.

You may also wish to integrate it with the student registration system of Project 21.3 (perhaps being implemented by another project team).

Project 21.5 Design and implement a Web-based system for booking classrooms at your university. Periodic booking (fixed days/times each week for a whole semester) must be supported. Cancellation of specific lectures in a periodic booking should also be supported.

You may also wish to integrate it with the student registration system of Project 21.3 (perhaps being implemented by another project team) so that classrooms can be booked for courses, and cancellations of a lecture or extra lectures can be noted at a single interface, and will be reflected in the classroom booking and communicated to students via e-mail.

Project 21.6 Design and implement a system for managing online multiple-choice tests. You should support distributed contribution of questions (by teaching assistants, for example), editing of questions by whoever is in charge of the course, and creation of tests from the available set of questions. You should also be able to administer tests online, either at a fixed time for all students, or at any time but with a time limit from start to finish (support one or both), and give students feedback on their scores at the end of the allotted time.

Project 21.7 Design and implement a system for managing e-mail customer service. Incoming mail goes to a common pool. There is a set of customer service agents who reply to e-mail. If the e-mail is part of an ongoing series of replies (tracked using the in-reply-to field of e-mail) the mail should preferably be replied to by the same agent who replied earlier. The system should track all incoming mail and replies, so an agent can see the history of questions from a customer before replying to an email.

Project 21.8 Design and implement a simple electronic marketplace where items can be listed for sale or for purchase under various categories (which should form a hierarchy). You may also wish to support alerting services, whereby a user can register interest in items in a particular category, perhaps with other constraints as well, without publicly advertising his/her interest, and is notified when such an item is listed for sale.

Project 21.9 Design and implement a Web-based newsgroup system. Users should be able to subscribe to newsgroups, and browse articles in newsgroups. The system tracks which articles were read by a user, so they are not displayed again. Also provide search against old articles.

You may also wish to provide a rating service for articles, so that articles with high rating are highlighted permitting the busy reader to skip low-rated articles.

Project 21.10 Design and implement a Web-based system for managing a sports "ladder." Many people register, and may be given some initial rankings (perhaps based on past performance). Anyone can challenge anyone else to a match, and the rankings are adjusted according to the result.

One simple system for adjusting rankings just moves the winner ahead of the loser in the rank order, in case the winner was behind earlier. You can try to invent more complicated rank adjustment systems.

Project 21.11 Design and implement a publications listing service. The service should permit entering of information about publications, such as title, authors, year, where the publication appeared, pages, and so forth. Authors should be a separate entity with attributes such as name, institution, department, e-mail, address, and home page.

Your application should support multiple views on the same data. For instance, you should provide all publications by a given author (sorted by year, for example), or all publications by authors from a given institution or department. You should also support search by keywords, on the overall database as well as within each of the views.

Bibliographical Notes

Information about servlets, including tutorials, standard specifications, and software, is available on java.sun.com/products/servlet. Information about JSP is available at java.sun.com/products/jsp.

An early proposal for a database-system benchmark (the Wisconsin benchmark) was made by Bitton et al. [1983]. The TPC-A,-B, and -C benchmarks are described in Gray [1991]. An online version of all the TPC benchmarks descriptions, as well as benchmark results, is available on the World Wide Web at the URL www.tpc.org; the site also contains up-to-date information about new benchmark proposals. Poess and Floyd [2000] gives an overview of the TPC-H, TPC-R, and TPC-W benchmarks. The OO1 benchmark for OODBs is described in Cattell and Skeen [1992]; the OO7 benchmark is described in Carey et al. [1993].

Kleinrock [1975] and Kleinrock [1976] is a popular two-volume textbook on queueing theory.

Shasha [1992] provides a good overview of database tuning. O'Neil and O'Neil [2000] provides a very good textbook coverage of performance measurement and tuning. The five minute and one minute rules are described in Gray and Putzolu [1987] and Gray and Graefe [1997]. Brown et al. [1994] describes an approach to automated tuning. Index selection and materialized view selection are addressed by Ross et al. [1996], Labio et al. [1997], Gupta [1997], Chaudhuri and Narasayya [1997], Agrawal et al. [2000] and Mistry et al. [2001].

The American National Standard SQL-86 is described in ANSI [1986]. The IBM Systems Application Architecture definition of SQL is specified by IBM [1987]. The standards for SQL-89 and SQL-92 are available as ANSI [1989] and ANSI [1992] respectively. For references on the SQL:1999 standard, see the bibliographical notes of Chapter 9.

The X/Open SQL call-level interface is defined in X/Open [1993]; the ODBC API is described in Microsoft [1997] and Sanders [1998]. The X/Open XA interface is defined in X/Open [1991]. More information about ODBC, OLE-DB, and ADO can be found on the Web site www.microsoft.com/data, and in a number of books on the subject that can be found through www.amazon.com. The ODMG 3.0 standard is defined in Cattell [2000]. *ACM Sigmod Record*, which is published quarterly, has a regular section on standards in databases, including benchmark standards.

A wealth of information on XML based standards is available online. You can use a Web search engine such as Google to search for more detailed and up-to-date information about the XML and other standards.

Loeb [1998] provides a detailed description of secure electronic transactions. Business process reengineering is covered by Cook [1996]. Kirchmer [1999] describes application implementation using standard software such as Enterprise Resource Planning (ERP) packages. Umar [1997] covers reengineering and issues in dealing with legacy systems.

Tools

There are many Web development tools that support database connectivity through servlets, JSP, Javascript, or other mechanisms. We list a few of the better-known ones here: Java SDK from Sun (java.sun.com), Apache's Tomcat (jakarta.apache.org) and Web server (apache.org), IBM WebSphere (www.software.ibm.com), Microsoft's ASP tools (www.microsoft.com), Allaire's Coldfusion and JRun products (www.allaire.com), Caucho's Resin (www.caucho.com), and Zope (www.zope.org). A few of these, such as Apache, are free for any use, some are free for noncommercial use or for personal use, while others need to be paid for. See the respective Web sites for more information.