P A R T 7

Other Topics

Chapter 21 covers a number of issues in building and maintaining applications and administering database systems. The chapter first outlines how to implement user interfaces, in particular Web-based interfaces. Other issues such as performance tuning (to improve application speed), standards issues in electronic commerce, and how to handle legacy systems are also covered in this chapter.

Chapter 22 describes a number of recent advances in querying and information retrieval. It first covers SQL extensions to support new types of queries, in particular queries typically posed by data analysts. It next covers data warehousing, whereby data generated by different parts of an organization are gathered centrally. The chapter then outlines data mining, which aims at finding patterns of various complex forms in large volumes of data. Finally, the chapter describes information retrieval, which deals with techniques for querying collections of text documents, such as Web pages, to find documents of interest.

Chapter 22 describes data types, such as temporal data, spatial data, and multimedia data, and the issues in storing such data in databases. Applications such as mobile computing and its connections with databases, are also described in this chapter.

Finally, Chapter 23 describes several advanced transaction-processing techniques, including transaction-processing monitors, transactional workflows, long-duration transactions, and multidatabase transactions.

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

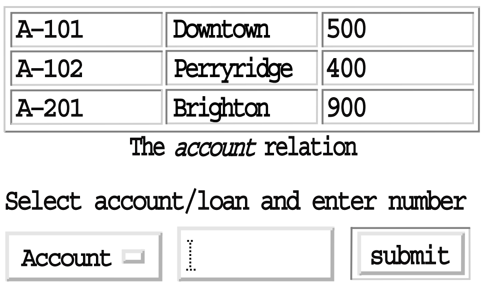
```
<html>
<body>
<table BORDER COLS=3>
<tr> <td>A-101</td> <td>Downtown</td> <td>500</td> </tr>
<tr> <td>A-102</td> <td>Perryridge</td> <td>400</td> </tr>
<tr> <td>A-201</td> <td>Brighton   </td> <td>900</td> </tr>
</table>
<center> The <i>account</i> relation </center>

<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.

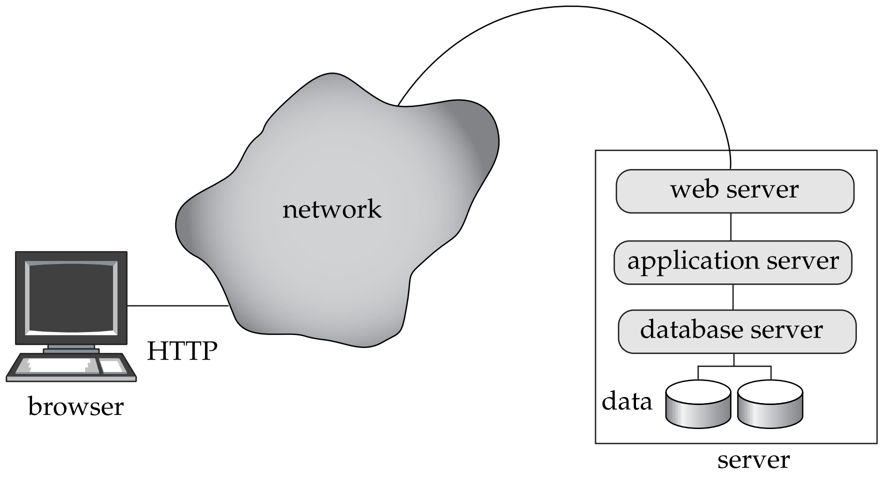


Figure 21.3 Three-tier Web architecture.

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

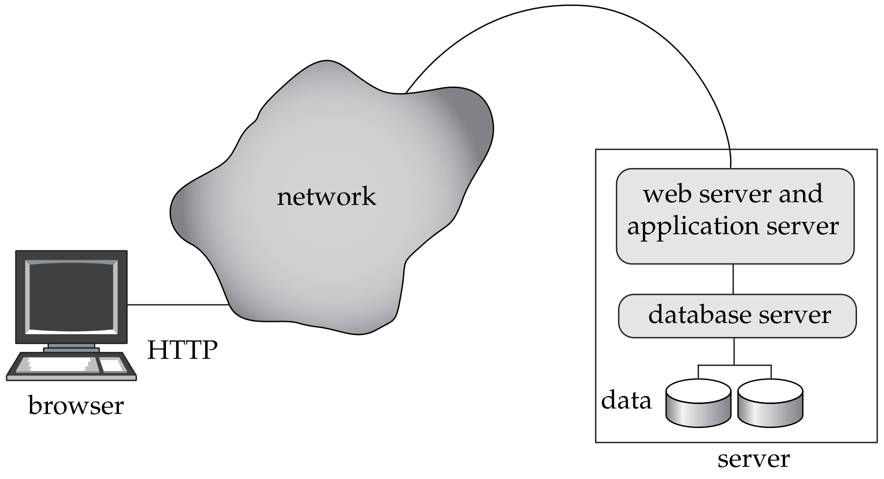


Figure 21.4 Two-tier Web architecture.

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.

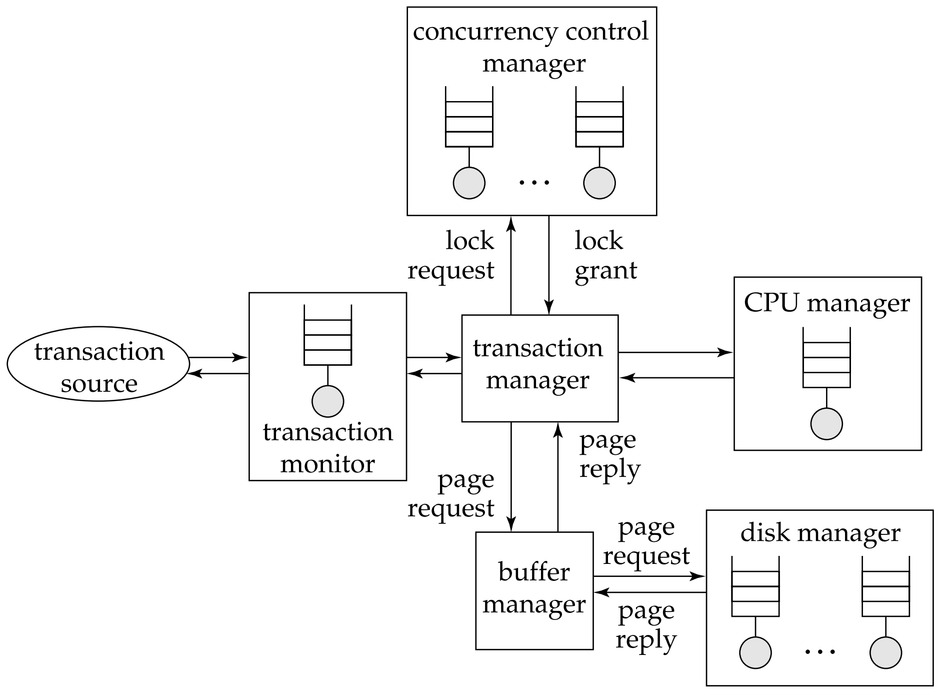


Figure 21.6 Queues in a database system.

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.

CHAPTER 22

Advanced Querying and Information Retrieval

Businesses have begun to exploit the burgeoning data online to make better decisions about their activities, such as what items to stock and how best to target customers to increase sales. Many of their queries are rather complicated, however, and certain types of information cannot be extracted even by using SQL.

Several techniques and tools are available to help with decision support. Several tools for data analysis allow analysts to view data in different ways. Other analysis tools precompute summaries of very large amounts of data, in order to give fast responses to queries. The SQL:1999 standard now contains additional constructs to support data analysis. Another approach to getting knowledge from data is to use *data mining*, which aims at detecting various types of patterns in large volumes of data. Data mining supplements various types of statistical techniques with similar goals.

Textual data, too, has grown explosively. Textual data is unstructured, unlike the rigidly structured data in relational databases. Querying of unstructured textual data is referred to as *information retrieval*. Information retrieval systems have much in common with database systems—in particular, the storage and retrieval of data on secondary storage. However, the emphasis in the field of information systems is different from that in database systems, concentrating on issues such as querying based on keywords; the relevance of documents to the query; and the analysis, classification, and indexing of documents.

This chapter covers decision support, including online analytical processing and data mining and information retrieval.

22.1 Decision-Support Systems

Database applications can be broadly classified into transaction processing and decision support, as we have seen earlier in Section 21.3.2. Transaction-processing systems are widely used today, and companies have accumulated a vast amount of information generated by these systems.

For example, company databases often contain enormous quantities of information about customers and transactions. The size of the information storage required may range up to hundreds of gigabytes, or even terabytes, for large retail chains. Transaction information for a retailer may include the name or identifier (such as credit-card number) of the customer, the items purchased, the price paid, and the dates on which the purchases were made. Information about the items purchased may include the name of the item, the manufacturer, the model number, the color, and the size. Customer information may include credit history, annual income, residence, age, and even educational background.

Such large databases can be treasure troves of information for making business decisions, such as what items to stock and what discounts to offer. For instance, a retail company may notice a sudden spurt in purchases of flannel shirts in the Pacific Northwest, may realize that there is a trend, and may start stocking a larger number of such shirts in shops in that area. As another example, a car company may find, on querying its database, that most of its small sports cars are bought by young women whose annual incomes are above $50,000. The company may then target its marketing to attract more such women to buy its small sports cars, and may avoid wasting money trying to attract other categories of people to buy those cars. In both cases, the company has identified patterns in customer behavior, and has used the patterns to make business decisions.

The storage and retrieval of data for decision support raises several issues:

- Although many decision support queries can be written in SQL, others either cannot be expressed in SQL or cannot be expressed easily in SQL. Several SQL extensions have therefore been proposed to make data analysis easier. The area of *online analytical processing* (OLAP) deals with tools and techniques for data analysis that can give nearly instantaneous answers to queries requesting summarized data, even though the database may be extremely large. In Section 22.2, we study SQL extensions for data analysis, and techniques for online analytical processing.

- Database query languages are not suited to the performance of detailed **statistical analyses** of data. There are several packages, such as SAS and S++, that help in statistical analysis. Such packages have been interfaced with databases, to allow large volumes of data to be stored in the database and retrieved efficiently for analysis. The field of statistical analysis is a large discipline on its own; see the references in the bibliographical notes for more information.

- Knowledge-discovery techniques attempt to discover automatically statistical rules and patterns from data. The field of *data mining* combines knowledge discovery techniques invented by artificial intelligence researchers and statistical analysts, with efficient implementation techniques that enable them to be used on extremely large databases. Section 22.3 discusses data mining.

- Large companies have diverse sources of data that they need to use for making business decisions. The sources may store the data under different schemas. For performance reasons (as well as for reasons of organization control), the data sources usually will not permit other parts of the company to retrieve data on demand.

To execute queries efficiently on such diverse data, companies have built *data warehouses*. Data warehouses gather data from multiple sources under a unified schema, at a single site. Thus, they provide the user a single uniform interface to data. We study issues in building and maintaining a data warehouse in Section 22.4.

The area of **decision support** can be broadly viewed as covering all the above areas, although some people use the term in a narrower sense that excludes statistical analysis and data mining.

22.2 Data Analysis and OLAP

Although complex statistical analysis is best left to statistics packages, databases should support simple, commonly used, forms of data analysis. Since the data stored in databases are usually large in volume, they need to be summarized in some fashion if we are to derive information that humans can use.

OLAP tools support interactive analysis of summary information. Several SQL extensions have been developed to support OLAP tools. There are many commonly used tasks that cannot be done using the basic SQL aggregation and grouping facilities. Examples include finding percentiles, or cumulative distributions, or aggregates over sliding windows on sequentially ordered data. A number of extensions of SQL have been recently proposed to support such tasks, and implemented in products such as Oracle and IBM DB2.

22.2.1 Online Analytical Processing

Statistical analysis often requires grouping on multiple attributes. Consider an application where a shop wants to find out what kinds of clothes are popular. Let us suppose that clothes are characterized by their item-name, color, and size, and that we have a relation *sales* with the schema *sales*(*item-name, color, size, number*). Suppose that *item-name* can take on the values (skirt, dress, shirt, pant), *color* can take on the values (dark, pastel, white), and *size* can take on values (small, medium, large).

Given a relation used for data analysis, we can identify some of its attributes as **measure** attributes, since they measure some value, and can be aggregated upon. For instance, the attribute *number* of the *sales* relation is a measure attribute, since it measures the number of units sold. Some (or all) of the other attributes of the relation are identified as **dimension attributes**, since they define the dimensions on which measure attributes, and summaries of measure attributes, are viewed. In the *sales* relation, *item-name, color*, and *size* are dimension attributes. (A more realistic version of the *sales* relation would have additional dimensions, such as time and sales location, and additional measures such as monetary value of the sale.)

Data that can be modeled as dimension attributes and measure attributes are called **multidimensional data**.

size: **all**

item-name \ *color*	dark	pastel	white	Total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

Figure 22.1 Cross tabulation of *sales* by *item-name* and *color*.

To analyze the multidimensional data, a manager may want to see data laid out as shown in the table in Figure 22.1. The table shows total numbers for different combinations of *item-name* and *color*. The value of *size* is specified to be **all**, indicating that the displayed values are a summary across all values of *size*.

The table in Figure 22.1 is an example of a **cross-tabulation** (or **cross-tab**, for short), also referred to as a **pivot-table**. In general, a cross-tab is a table where values for one attribute (say A) form the row headers, values for another attribute (say B) form the column headers, and the values in an individual cell are derived as follows. Each cell can be identified by (a_i, b_j), where a_i is a value for A and b_j a value for B. If there is at most one tuple with any (a_i, b_j) value, the value in the cell is derived from that single tuple (if any); for instance, it could be the value of one or more other attributes of the tuple. If there can be multiple tuples with an (a_i, b_j) value, the value in the cell must be derived by aggregation on the tuples with that value. In our example, the aggregation used is the sum of the values for attribute *number*. In our example, the cross-tab also has an extra column and an extra row storing the totals of the cells in the row/column. Most cross-tabs have such summary rows and columns.

A cross-tab is different from relational tables usually stored in databases, since the number of columns in the cross-tab depends on the actual data. A change in the data values may result in adding more columns, which is not desirable for data storage. However, a cross-tab view is desirable for display to users. It is straightforward to represent a cross-tab without summary values in a relational form with a fixed number of columns. A cross-tab with summary rows/columns can be represented by introducing a special value **all** to represent subtotals, as in Figure 22.2. The SQL:1999 standard actually uses the **null** value in place of **all**, but to avoid confusion with regular null values, we shall continue to use **all**.

Consider the tuples (skirt, **all**, 53) and (dress, **all**, 35). We have obtained these tuples by eliminating individual tuples with different values for *color*, and by replacing the value of *number* by an aggregate—namely, sum. The value **all** can be thought of as representing the set of all values for an attribute. Tuples with the value **all** only for the *color* dimension can be obtained by an SQL query performing a group by on the column *item-name*. Similarly, a group by on *color* can be used to get the tuples with the value **all** for *item-name*, and a group by with no attributes (which can simply be omitted in SQL) can be used to get the tuple with value **all** for *item-name* and *color*.

item-name	*color*	*number*
skirt	dark	8
skirt	pastel	35
skirt	white	10
skirt	**all**	53
dress	dark	20
dress	pastel	10
dress	white	5
dress	**all**	35
shirt	dark	14
shirt	pastel	7
shirt	white	28
shirt	**all**	49
pant	dark	20
pant	pastel	2
pant	white	5
pant	**all**	27
all	dark	62
all	pastel	54
all	white	48
all	**all**	164

Figure 22.2 Relational representation of the data in Figure 22.1.

The generalization of a cross-tab, which is 2-dimensional, to n dimensions can be visualized as an n-dimensional cube, called the **data cube**. Figure 22.3 shows a data cube on the *sales* relation. The data cube has three dimensions, namely *item-name*, *color*, and *size*, and the measure attribute is *number*. Each cell is identified by values for these three dimensions. Each cell in the data cube contains a value, just as in a cross-tab. In Figure 22.3, the value contained in a cell is shown on one of the faces of the cell; other faces of the cell are shown blank if they are visible.

The value for a dimension may be **all**, in which case the cell contains a summary over all values of that dimension, as in the case of cross-tabs. The number of different ways in which the tuples can be grouped for aggregation can be large. In fact, for a table with n dimensions, aggregation can be performed with grouping on each of the 2^n subsets of the n dimensions.[1]

An online analytical processing or OLAP system is an interactive system that permits an analyst to view different summaries of multidimensional data. The word *online* indicates that the an analyst must be able to request new summaries and get responses online, within a few seconds, and should not be forced to wait for a long time to see the result of a query.

With an OLAP system, a data analyst can look at different cross-tabs on the same data by interactively selecting the attributes in the cross-tab. Each cross-tab is a

1. Grouping on the set of all n dimensions is useful only if the table may have duplicates.

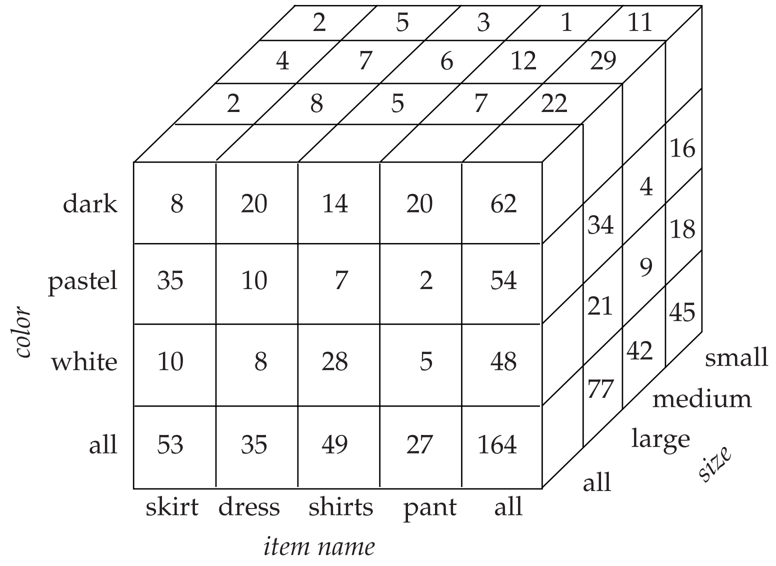


Figure 22.3 Three-dimensional data cube.

two-dimensional view on a multidimensional data cube. For instance the analyst may select a cross-tab on *item-name* and *size*, or a cross-tab on *color* and *size*. The operation of changing the dimensions used in a cross-tab is called **pivoting**.

An OLAP system provides other functionality as well. For instance, the analyst may wish to see a cross-tab on *item-name* and *color* for a fixed value of *size*, for example, large, instead of the sum across all sizes. Such an operation is referred to as **slicing**, since it can be thought of as viewing a slice of the data cube. The operation is sometimes called **dicing**, particularly when values for multiple dimensions are fixed.

When a cross-tab is used to view a multidimensional cube, the values of dimension attributes that are not part of the cross-tab are shown above the cross-tab. The value of such an attribute can be **all**, as shown in Figure 22.1, indicating that data in the cross-tab are a summary over all values for the attribute. Slicing/dicing simply consists of selecting specific values for these attributes, which are then displayed on top of the cross-tab.

OLAP systems permit users to view data at any desired level of granularity. The operation of moving from finer-granularity data to a coarser granularity (by means of aggregation) is called a **rollup**. In our example, starting from the data cube on the *sales* table, we got our example cross-tab by rolling up on the attribute *size*. The opposite operation—that of moving from coarser-granularity data to finer-granularity data—is called a **drill down**. Clearly, finer-granularity data cannot be generated from coarse-granularity data; they must be generated either from the original data, or from even finer-granularity summary data.

Analysts may wish to view a dimension at different levels of detail. For instance, an attribute of type **datetime** contains a date and a time of day. Using time precise to a second (or less) may not be meaningful: An analyst who is interested in rough time

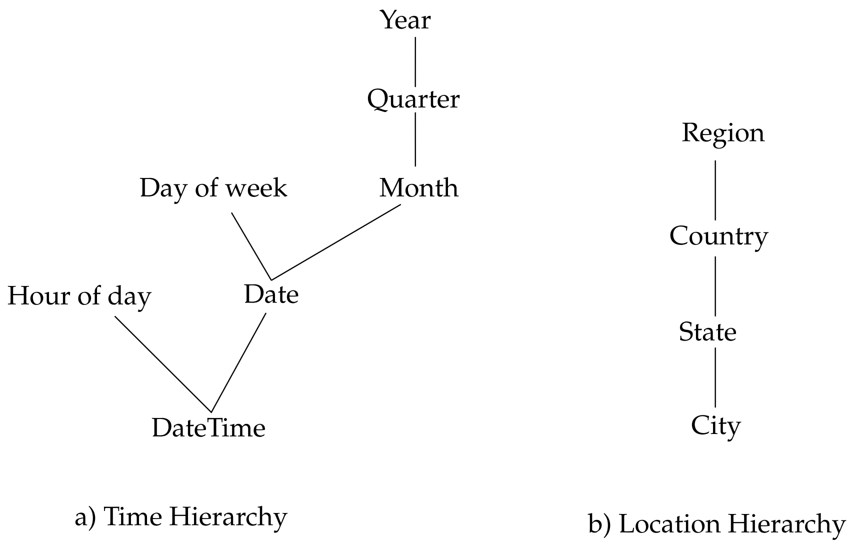


Figure 22.4 Hierarchies on dimensions.

of day may look at only the hour value. An analyst who is interested in sales by day of the week may map the date to a day-of-the-week and look only at that. Another analyst may be interested in aggregates over a month, or a quarter, or for an entire year.

The different levels of detail for an attribute can be organized into a **hierarchy**. Figure 22.4(a) shows a hierarchy on the **datetime** attribute. As another example, Figure 22.4(b) shows a hierarchy on location, with the city being at the bottom of the hierarchy, state above it, country at the next level, and region being the top level. In our earlier example, clothes can be grouped by category (for instance, menswear or womenswear); *category* would then lie above *item-name* in our hierarchy on clothes. At the level of actual values, skirts and dresses would fall under the womenswear category and pants and shirts under the menswear category.

An analyst may be interested in viewing sales of clothes divided as menswear and womenswear, and not interested in individual values. After viewing the aggregates at the level of womenswear and menswear, an analyst may *drill down the hierarchy* to look at individual values. An analyst looking at the detailed level may *drill up the hierarchy*, and look at coarser-level aggregates. Both levels can be displayed on the same cross-tab, as in Figure 22.5.

22.2.2 OLAP Implementation

The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems. Later, OLAP facilities were integrated into relational systems, with data stored in a relational database. Such systems are referred to as **relational OLAP (ROLAP)** systems. Hybrid

category	*item-name*	dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	skirt	14	14	28	49	
	dress	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Figure 22.5 Cross tabulation of *sales* with hierarchy on *item-name*.

systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems.

Many OLAP systems are implemented as client–server systems. The server contains the relational database as well as any MOLAP data cubes. Client systems obtain views of the data by communicating with the server.

A naïve way of computing the entire data cube (all groupings) on a relation is to use any standard algorithm for computing aggregate operations, one grouping at a time. The naïve algorithm would require a large number of scans of the relation. A simple optimization is to compute an aggregation on, say, (*item-name*, *color*) from an aggregation (*item-name*, *color*, *size*), instead of from the original relation. For the standard SQL aggregate functions, we can compute an aggregate with grouping on a set of attributes A from an aggregate with grouping on a set of attributes B if $A \subseteq B$; you can do so as an exercise (see Exercise 22.1), but note that to compute **avg**, we additionally need the **count** value. (For some non-standard aggregate functions, such as median, aggregates cannot be computed as above; the optimization described here do not apply to such "*non-decomposable*" aggregate functions.) The amount of data read drops significantly by computing an aggregate from another aggregate, instead of from the original relation. Further improvements are possible; for instance, multiple groupings can be computed on a single scan of the data. See the bibliographical notes for references to algorithms for efficiently computing data cubes.

Early OLAP implementations precomputed and stored entire data cubes, that is, groupings on all subsets of the dimension attributes. Precomputation allows OLAP queries to be answered within a few seconds, even on datasets that may contain millions of tuples adding up to gigabytes of data. However, there are 2^n groupings with n dimension attributes; hierarchies on attributes increase the number further. As a result, the entire data cube is often larger than the original relation that formed the data cube and in many cases it is not feasible to store the entire data cube.

Instead of precomputing and storing all possible groupings, it makes sense to precompute and store some of the groupings, and to compute others on demand. Instead of computing queries from the original relation, which may take a very long time, we can compute them from other precomputed queries. For instance, suppose a query requires summaries by (*item-name*, *color*), which has not been precomputed. The query result can be computed from summaries by (*item-name*, *color*, *size*), if that has been precomputed. See the bibliographical notes for references on how to select a good set of groupings for precomputation, given limits on the storage available for precomputed results.

The data in a data cube cannot be generated by a single SQL query using the basic **group by** constructs, since aggregates are computed for several different groupings of the dimension attributes. Section 22.2.3 discusses SQL extensions to support OLAP functionality.

22.2.3 Extended Aggregation

The SQL-92 aggregation functionality is limited, so several extensions were implemented by different databases. The SQL:1999 standard, however, defines a rich set of aggregate functions, which we outline in this section and in the next two sections. The Oracle and IBM DB2 databases support most of these features, and other databases will no doubt support these features in the near future.

The new aggregate functions on single attributes are standard deviation and variance (**stddev** and **variance**). Standard deviation is the square root of variance.[2] Some database systems support other aggregate functions such as median and mode. Some database systems even allow users to add new aggregate functions.

SQL:1999 also supports a new class of **binary aggregate functions**, which can compute statistical results on pairs of attributes; they include correlations, covariances, and regression curves, which give a line approximating the relation between the values of the pair of attributes. Definitions of these functions may be found in any standard textbook on statistics, such as those referenced in the bibliographical notes.

SQL:1999 also supports generalizations of the **group by** construct, using the **cube** and **rollup** constructs. A representative use of the **cube** construct is:

select *item-name, color, size*, **sum**(*number*)
from *sales*
group by cube(*item-name, color, size*)

This query computes the union of eight different groupings of the *sales* relation:

{ (*item-name, color, size*), (*item-name, color*), (*item-name, size*),
(*color, size*), (*item-name*), (*color*), (*size*), () }

where () denotes an empty **group by** list.

For each grouping, the result contains the null value for attributes not present in the grouping. For instance, the table in Figure 22.2, with occurrences of **all** replaced by *null*, can be computed by the query

select *item-name, color*, **sum**(*number*)
from *sales*
group by cube(*item-name, color*)

2. The SQL:1999 standard actually supports two types of variance, called population variance and sample variance, and correspondingly two types of standard deviation. The definitions of the two types differ slightly; see a statistics textbook for details.

A representative **rollup** construct is

> **select** *item-name, color, size*, **sum**(*number*)
> **from** *sales*
> **group by rollup**(*item-name, color, size*)

Here, only four groupings are generated:

{ (*item-name, color, size*), (*item-name, color*), (*item-name*), () }

Rollup can be used to generate aggregates at multiple levels of a hierarchy on a column. For instance, suppose we have a table *itemcategory(item-name, category)* giving the category of each item. Then the query

> **select** *category, item-name*, **sum**(*number*)
> **from** *sales, category*
> **where** *sales.item-name = itemcategory.item-name*
> **group by rollup**(*category, item-name*)

would give a hierarchical summary by *item-name* and by *category*.

Multiple **rollup**s and **cube**s can be used in a single group by clause. For instance, the following query

> **select** *item-name, color, size*, **sum**(*number*)
> **from** *sales*
> **group by rollup**(*item-name*), **rollup**(*color, size*)

generates the groupings

{ (*item-name, color, size*), (*item-name, color*), (*item-name*),
(*color, size*), (*color*), () }

To understand why, note that **rollup**(*item-name*) generates two groupings, {(*item-name*), ()}, and **rollup**(*color, size*) generates three groupings, {(*color, size*), (*color*), () }. The cross product of the two gives us the six groupings shown.

As we mentioned in Section 22.2.1, SQL:1999 uses the value **null** to indicate the usual sense of null as well as **all**. This dual use of **null** can cause ambiguity if the attributes used in a rollup or cube clause contain null values. The function **grouping** can be applied on an attribute; it returns 1 if the value is a null value representing **all**, and returns 0 in all other cases. Consider the following query:

> **select** *item-name, color, size*, **sum**(*number*),
>   **grouping**(*item-name*) **as** *item-name-flag*,
>   **grouping**(*color*) **as** *color-flag*,
>   **grouping**(*size*) **as** *size-flag*
> **from** *sales*
> **group by cube**(*item-name, color, size*)

The output is the same as in the version of the query without **grouping**, but with three extra columns called *item-name-flag, color-flag*, and *size-flag*. In each tuple, the value of a flag field is 1 if the corresponding field is a null representing **all**.

Instead of using tags to indicate nulls that represent **all**, we can replace the null value by a value of our choice:

decode(**grouping**(*item-name*), 1, 'all', *item-name*)

This expression returns the value "all" if the value of *item-name* is a null corresponding to **all**, and returns the actual value of *item-name* otherwise. This expression can be used in place of *item-name* in the select clause to get "all" in the output of the query, in place of nulls representing **all**.

Neither the **rollup** nor the **cube** clause gives complete control on the groupings that are generated. For instance, we cannot use them to specify that we want only groupings {(*color, size*), (*size, item-name*)}. Such restricted groupings can be generated by using the **grouping** construct in the **having** clause; we leave the details as an exercise for you.

22.2.4 Ranking

Finding the position of a value in a larger set is a common operation. For instance, we may wish to assign students a rank in class based on their total marks, with the rank 1 going to the student with the highest marks, the rank 2 to the student with the next highest marks, and so on. While such queries can be expressed in SQL-92, they are difficult to express and inefficient to evaluate. Programmers often resort to writing the query partly in SQL and partly in a programming language. A related type of query is to find the percentile in which a value in a (multi)set belongs, for example, the bottom third, middle third, or top third. We study SQL:1999 support for these types of queries here.

Ranking is done in conjunction with an **order by** specification. Suppose we are given a relation *student-marks(student-id, marks)* which stores the marks obtained by each student. The following query gives the rank of each student.

```
select student-id, rank() over (order by (marks) desc) as s-rank
from student-marks
```

Note that the order of tuples in the output is not defined, so they may not be sorted by rank. An extra **order by** clause is needed to get them in sorted order, as shown below.

```
select student-id, rank () over (order by (marks) desc) as s-rank
from student-marks order by s-rank
```

A basic issue with ranking is how to deal with the case of multiple tuples that are the same on the ordering attribute(s). In our example, this means deciding what to do if there are two students with the same marks. The **rank** function gives the same rank to all tuples that are equal on the **order by** attributes. For instance, if the highest mark is shared by two students, both would get rank 1. The next rank given would be 3, not 2, so if three students get the next highest mark, they would all get rank 3, and the next student(s) would get rank 5, and so on. There is also a **dense_rank** function that does not create gaps in the ordering. In the above example, the tuples with the second highest value all get rank 2, and tuples with the third highest value get rank 3, and so on.

Ranking can be done within partitions of the data. For instance, suppose we have an additional relation *student-section(student-id, section)* that stores for each student the section in which the student studies. The following query then gives the rank of students within each section.

```
select student-id, section,
    rank () over (partition by section order by marks desc) as sec-rank
from student-marks, student-section
where student-marks.student-id = student-section.student-id
order by section, sec-rank
```

The outer **order by** clause orders the result tuples by section, and within each section by the rank.

Multiple **rank** expressions can be used within a single select statement; thus we can obtain the overall rank and the rank within the section by using two **rank** expressions in the same **select** clause. An interesting question is what happens when ranking (possibly with partitioning) occurs along with a **group by** clause. In this case, the **group by** clause is applied first, and partitioning and ranking are done on the results of the group by. Thus aggregate values can then be used for ranking. For example, suppose we had marks for each student for each of several subjects. To rank students by the sum of their marks in different subjects, we can use a **group by** clause to compute the aggregate marks for each student, and then rank students by the aggregate sum. We leave details as an exercise for you.

The ranking functions can be used to find the top n tuples by embedding a ranking query within an outer-level query; we leave details as an exercise. Note that bottom n is simply the same as top n with a reverse sorting order. Several database systems provide nonstandard SQL extensions to specify directly that only the top n results are required; such extensions do not require the rank function, and simplify the job of the optimizer, but are (currently) not as general since they do not support partitioning.

SQL:1999 also specifies several other functions that can be used in place of **rank**. For instance, **percent_rank** of a tuple gives the rank of the tuple as a fraction. If there are n tuples in the partition[3] and the rank of the tuple is r, then its percent rank is defined as $(r-1)/(n-1)$ (and as null if there is only one tuple in the partition). The function **cume_dist**, short for cumulative distribution, for a tuple is defined as p/n where p is the number of tuples in the partition with ordering values preceding or equal to the ordering value of the tuple, and n is the number of tuples in the parti-

3. The entire set is treated as a single partition if no explicit partition is used.

tion. The function **row_number** sorts the rows and gives each row a unique number corresponding to its position in the sort order; different rows with the same ordering value would get different row numbers, in a nondeterministic fashion.

Finally, for a given constant n, the ranking function **ntile**(n) takes the tuples in each partition in the specified order, and divides them into n buckets with equal numbers of tuples.[4] For each tuple, **ntile**(n) then gives the number of the bucket in which it is placed, with bucket numbers starting with 1. This function is particularly useful for constructing histograms based on percentiles. For instance, we can sort employees by salary, and use **ntile**(3) to find which range (bottom third, middle third, or top third) each employee is in, and compute the total salary earned by employees in each range:

```
select threetile, sum(salary)
from (
    select salary, ntile(3) over (order by (salary)) as threetile
    from employee) as s
group by threetile.
```

The presence of null values can complicate the definition of rank, since it is not clear where they should occur first in the sort order. SQL:1999 permits the user to specify where they should occur by using **nulls first** or **nulls last**, for instance

```
select student-id, rank () over (order by marks desc nulls last) as s-rank
from student-marks
```

22.2.5 Windowing

An example of a *window* query is query that, given sales values for each date, calculates for each date the average of the sales on that day, the previous day, and the next day; such moving average queries are used to smooth out random variations. Another example of a window query is one that finds the cumulative balance in an account, given a relation specifying the deposits and withdrawals on an account. Such queries are either hard or impossible (depending on the exact query) to express in basic SQL.

SQL:1999 provides a windowing feature to support such queries. In contrast to **group by**, the same tuple can exist in multiple windows. Suppose we are given a relation *transaction*(*account-number, date-time, value*), where *value* is positive for a deposit and negative for a withdrawal. We assume there is at most one transaction per *date-time* value.

Consider the query

4. If the total number of tuples in a partition is not divisible by n, then the number of tuples in each bucket can differ by at most 1. Tuples with the same value for the ordering attribute may be assigned to different buckets, nondeterministically, in order to make the number of tuples in each bucket equal.

```
select account-number, date-time,
    sum(value) over
        (partition by account-number
         order by date-time
         rows unbounded preceding)
    as balance
from transaction
order by account-number, date-time
```

The query gives the cumulative balances on each account just before each transaction on the account; the cumulative balance of the account is the sum of values of all earlier transactions on the account.

The **partition by** clause partitions tuples by account number, so for each row only the tuples in its partition are considered. A window is created for each tuple; the keywords **rows unbounded preceding** specify that the window for each tuple consists of all tuples in the partition that precede it in the specified order (here, increasing order of *date-time*). The aggregate function **sum**(*value*) is applied on all the tuples in the window. Observe that the query does not use a **group by** clause, since there is an output tuple for each tuple in the *transaction* relation.

While the query could be written without these extended constructs, it would be rather difficult to formulate. Note also that different windows can overlap, that is, a tuple may be present in more than one window.

Other types of windows can be specified. For instance, to get a window containing the previous 10 rows for each row, we can specify **rows** 10 **preceding**. To get a window containing the current, previous, and following row, we can use **between rows** 1 **preceding and** 1 **following**. To get the previous rows and the current row, we can say **between rows unbounded preceding and current**. Note that if the ordering is on a nonkey attribute, the result is not deterministic, since the order of tuples is not fully defined.

We can even specify windows by ranges of values, instead of numbers of rows. For instance, suppose the ordering value of a tuple is v; then **range between** 10 **preceding and current row** would give tuples whose ordering value is between $v - 10$ and v (both values inclusive). When dealing with dates, we can use **range interval** 10 **day preceding** to get a window containing tuples within the previous 10 days, but not including the date of the tuple.

Clearly, the windowing functionality of SQL:1999 is very rich and can be used to write rather complex queries with a small amount of effort.

22.3 Data Mining

The term **data mining** refers loosely to the process of semiautomatically analyzing large databases to find useful patterns. Like knowledge discovery in artificial intelligence (also called machine learning), or statistical analysis, data mining attempts to discover rules and patterns from data. However, data mining differs from machine learning and statistics in that it deals with large volumes of data, stored primarily on disk. That is, data mining deals with "knowledge discovery in databases."

Some types of knowledge discovered from a database can be represented by a set of **rules**. The following is an example of a rule, stated informally: "Young women with annual incomes greater than $50,000 are the most likely people to buy small sports cars." Of course such rules are not universally true, and have degrees of "support" and "confidence," as we shall see. Other types of knowledge are represented by equations relating different variables to each other, or by other mechanisms for predicting outcomes when the values of some variables are known.

There are a variety of possible types of patterns that may be useful, and different techniques are used to find different types of patterns. We shall study a few examples of patterns and see how they may be automatically derived from a database.

Usually there is a manual component to data mining, consisting of preprocessing data to a form acceptable to the algorithms, and postprocessing of discovered patterns to find novel ones that could be useful. There may also be more than one type of pattern that can be discovered from a given database, and manual interaction may be needed to pick useful types of patterns. For this reason, data mining is really a semiautomatic process in real life. However, in our description we concentrate on the automatic aspect of mining.

22.3.1 Applications of Data Mining

The discovered knowledge has numerous applications. The most widely used applications are those that require some sort of **prediction**. For instance, when a person applies for a credit card, the credit-card company wants to predict if the person is a good credit risk. The prediction is to be based on known attributes of the person, such as age, income, debts, and past debt repayment history. Rules for making the prediction are derived from the same attributes of past and current credit card holders, along with their observed behavior, such as whether they defaulted on their credit-card dues. Other types of prediction include predicting which customers may switch over to a competitor (these customers may be offered special discounts to tempt them not to switch), predicting which people are likely to respond to promotional mail ("junk mail"), or predicting what types of phone calling card usage are likely to be fraudulent.

Another class of applications looks for **associations**, for instance, books that tend to be bought together. If a customer buys a book, an online bookstore may suggest other associated books. If a person buys a camera, the system may suggest accessories that tend to be bought along with cameras. A good salesperson is aware of such patterns and exploits them to make additional sales. The challenge is to automate the process. Other types of associations may lead to discovery of causation. For instance, discovery of unexpected associations between a newly introduced medicine and cardiac problems led to the finding that the medicine may cause cardiac problems in some people. The medicine was then withdrawn from the market.

Associations are an example of **descriptive patterns**. **Clusters** are another example of such patterns. For example, over a century ago a cluster of typhoid cases was found around a well, which led to the discovery that the water in the well was contaminated and was spreading typhoid. Detection of clusters of disease remains important even today.

22.3.2 Classification

As mentioned in Section 22.3.1, prediction is one of the most important types of data mining. We outline what is classification, study techniques for building one type of classifiers, called decision tree classifiers, and then study other prediction techniques.

Abstractly, the **classification** problem is this: Given that items belong to one of several classes, and given past instances (called **training instances**) of items along with the classes to which they belong, the problem is to predict the class to which a new item belongs. The class of the new instance is not known, so other attributes of the instance must be used to predict the class.

Classification can be done by finding rules that partition the given data into disjoint groups. For instance, suppose that a credit-card company wants to decide whether or not to give a credit card to an applicant. The company has a variety of information about the person, such as her age, educational background, annual income, and current debts, that it can use for making a decision.

Some of this information could be relevant to the credit worthiness of the applicant, whereas some may not be. To make the decision, the company assigns a credit-worthiness level of excellent, good, average, or bad to each of a sample set of *current* customers according to each customer's payment history. Then, the company attempts to find rules that classify its current customers into excellent, good, average, or bad, on the basis of the information about the person, other than the actual payment history (which is unavailable for new customers). Let us consider just two attributes: education level (highest degree earned) and income. The rules may be of the following form:

$$\begin{aligned}&\forall person\ \ P,\ P.degree = masters \textbf{ and } P.income > 75,000\\&\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow\ P.credit = excellent\\&\forall\ person\ \ P,\ P.degree = bachelors \textbf{ or}\\&\qquad (P.income \geq 25,000 \textbf{ and } P.income \leq 75,000)\ \Rightarrow\ P.credit = good\end{aligned}$$

Similar rules would also be present for the other credit worthiness levels (average and bad).

The process of building a classifier starts from a sample of data, called a **training set**. For each tuple in the training set, the class to which the tuple belongs is already known. For instance, the training set for a credit-card application may be the existing customers, with their credit worthiness determined from their payment history. The actual data, or population, may consist of all people, including those who are not existing customers. There are several ways of building a classifier, as we shall see.

22.3.2.1 Decision Tree Classifiers

The decision tree classifier is a widely used technique for classification. As the name suggests, **decision tree classifiers** use a tree; each leaf node has an associated class, and each internal node has a predicate (or more generally, a function) associated with it. Figure 22.6 shows an example of a decision tree.

To classify a new instance, we start at the root, and traverse the tree to reach a leaf; at an internal node we evaluate the predicate (or function) on the data instance,

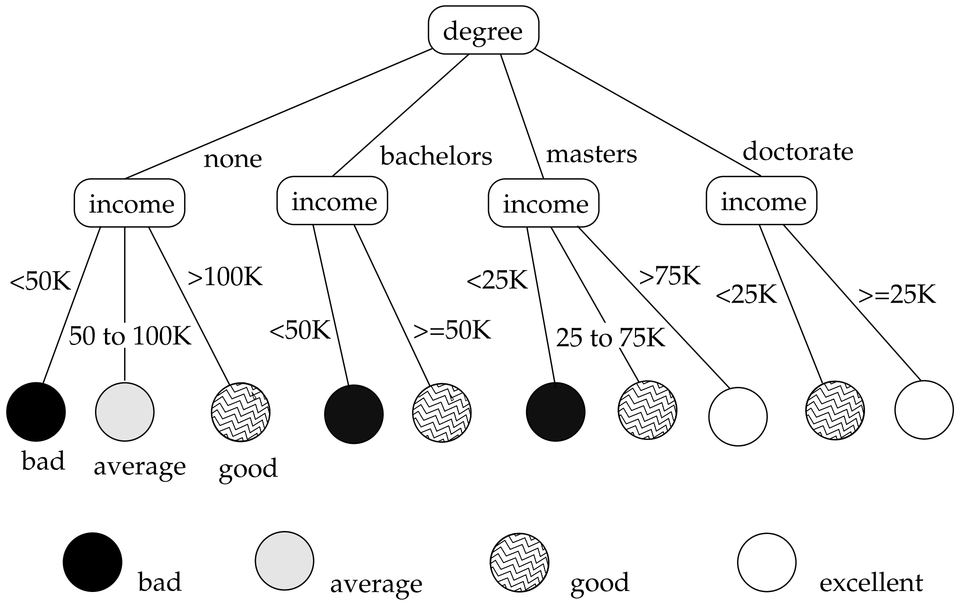


Figure 22.6 Classification tree.

to find which child to go to. The process continues till we reach a leaf node. For example, if the degree level of a person is masters, and the persons income is 40K, starting from the root we follow the edge labeled "masters," and from there the edge labeled "25K to 75K," to reach a leaf. The class at the leaf is "good," so we predict that the credit risk of that person is good.

Building Decision Tree Classifiers

The question then is how to build a decision tree classifier, given a set of training instances. The most common way of doing so is to use a **greedy** algorithm, which works recursively, starting at the root and building the tree downward. Initially there is only one node, the root, and all training instances are associated with that node.

At each node, if all, or "almost all" training instances associated with the node belong to the same class, then the node becomes a leaf node associated with that class. Otherwise, a **partitioning attribute** and **partitioning condition**s must be selected to create child nodes. The data associated with each child node is the set of training instances that satisfy the partitioning condition for that child node. In our example, the attribute *degree* is chosen, and four children, one for each value of degree, are created. The conditions for the four children nodes are *degree* = none, *degree* = bachelors, *degree* = masters, and *degree* = doctorate, respectively. The data associated with each child consist of training instances satisfying the condition associated with that child. At the node corresponding to masters, the attribute *income* is chosen, with the range of values partitioned into intervals 0 to 25,000, 25,000 to 50,000, 50,000 to 75,000, and over 75,000. The data associated with each node consist of training instances with the *degree* attribute being masters, and the *income* attribute being in each of these ranges, respectively. As an optimization, since the class for the range 25,000 to 50,000 and the range 50,000 to 75,000 is the same under the node *degree* = masters, the two ranges have been merged into a single range 25,000 to 75,000.

Best Splits

Intuitively, by choosing a sequence of partitioning attributes, we start with the set of all training instances, which is "impure" in the sense that it contains instances from many classes, and end up with leaves which are "pure" in the sense that at each leaf all training instances belong to only one class. We shall see shortly how to measure purity quantitatively. To judge the benefit of picking a particular attribute and condition for partitioning of the data at a node, we measure the purity of the data at the children resulting from partitioning by that attribute. The attribute and condition that result in the maximum purity are chosen.

The purity of a set S of training instances can be measured quantitatively in several ways. Suppose there are k classes, and of the instances in S the fraction of instances in class i is p_i. One measure of purity, the **Gini measure** is defined as

$$\text{Gini}(S) = 1 - \sum_{i-1}^{k} p_i^2$$

When all instances are in a single class, the Gini value is 0, while it reaches its maximum (of $1 - 1/k$) if each class has the same number of instances. Another measure of purity is the **entropy measure**, which is defined as

$$\text{Entropy}(S) = -\sum_{i-1}^{k} p_i \log_2 p_i$$

The entropy value is 0 if all instances are in a single class, and reaches its maximum when each class has the same number of instances. The entropy measure derives from information theory.

When a set S is split into multiple sets $S_i, i = 1, 2, \ldots, r$, we can measure the purity of the resultant set of sets as:

$$\text{Purity}(S_1, S_2, \ldots, S_r) = \sum_{i=1}^{r} \frac{|S_i|}{|S|} \text{purity}(S_i)$$

That is, the purity is the weighted average of the purity of the sets S_i. The above formula can be used with both the Gini measure and the entropy measure of purity.

The **information gain** due to a particular split of S into $S_i, i = 1, 2, \ldots, r$ is then

$$\text{Information-gain}(S, \{S_1, S_2, \ldots, S_r\}) = \text{purity}(S) - \text{purity}(S_1, S_2, \ldots, S_r)$$

Splits into fewer sets are preferable to splits into many sets, since they lead to simpler and more meaningful decision trees. The number of elements in each of the sets S_i may also be taken into account; otherwise, whether a set S_i has 0 elements or 1 element would make a big difference in the number of sets, although the split is the same for almost all the elements. The **information content** of a particular split can be defined in terms of entropy as

$$\text{Information-content}(S, \{S_1, S_2, \ldots, S_r\})) = -\sum_{i-1}^{r} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

All of this leads to a definition: The **best split** for an attribute is the one that gives the maximum **information gain ratio**, defined as

$$\frac{\text{Information-gain}(S, \{S_1, S_2, \ldots, S_r\})}{\text{Information-content}(S, \{S_1, S_2, \ldots, S_r\})}$$

Finding Best Splits

How do we find the best split for an attribute? How to split an attribute depends on the type of the attribute. Attributes can be either **continuous valued**, that is, the values can be ordered in a fashion meaningful to classification, such as age or income, or can be **categorical**, that is, they have no meaningful order, such as department names or country names. We do not expect the sort order of department names or country names to have any significance to classification.

Usually attributes that are numbers (integers/reals) are treated as continuous valued while character string attributes are treated as categorical, but this may be controlled by the user of the system. In our example, we have treated the attribute *degree* as categorical, and the attribute *income* as continuous valued.

We first consider how to find best splits for continuous-valued attributes. For simplicity, we shall only consider **binary splits** of continuous-valued attributes, that is, splits that result in two children. The case of **multiway splits** is more complicated; see the bibliographical notes for references on the subject.

To find the best binary split of a continuous-valued attribute, we first sort the attribute values in the training instances. We then compute the information gain obtained by splitting at each value. For example, if the training instances have values $1, 10, 15$, and 25 for an attribute, the split points considered are 1, 10, and 15; in each case values less than or equal to the split point form one partition and the rest of the values form the other partition. The best binary split for the attribute is the split that gives the maximum information gain.

For a categorical attribute, we can have a multiway split, with a child for each value of the attribute. This works fine for categorical attributes with only a few distinct values, such as degree or gender. However, if the attribute has many distinct values, such as department names in a large company, creating a child for each value is not a good idea. In such cases, we would try to combine multiple values into each child, to create a smaller number of children. See the bibliographical notes for references on how to do so.

Decision-Tree Construction Algorithm

The main idea of decision tree construction is to evaluate different attributes and different partitioning conditions, and pick the attribute and partitioning condition that results in the maximum information gain ratio. The same procedure works recur-

```
procedure GrowTree($S$)
    Partition($S$);

procedure Partition ($S$)
    if (purity($S$) $> \delta_p$ or $|S| < \delta_s$ ) then
        return;
    for each attribute $A$
        evaluate splits on attribute $A$;
    Use best split found (across all attributes) to partition
            $S$ into $S_1, S_2, \ldots, S_r$;
    for $i = 1, 2, \ldots, r$
        Partition($S_i$);
```

Figure 22.7 Recursive construction of a decision tree.

sively on each of the sets resulting from the split, thereby recursively constructing a decision tree. If the data can be perfectly classified, the recursion stops when the purity of a set is 0. However, often data are noisy, or a set may be so small that partitioning it further may not be justified statistically. In this case, the recursion stops when the purity of a set is "sufficiently high," and the class of resulting leaf is defined as the class of the majority of the elements of the set. In general, different branches of the tree could grow to different levels.

Figure 22.7 shows pseudocode for a recursive tree construction procedure, which takes a set of training instances S as parameter. The recursion stops when the set is sufficiently pure or the set S is too small for further partitioning to be statistically significant. The parameters δ_p and δ_s define cutoffs for purity and size; the system may give them default values, that may be overridden by users.

There are a wide variety of decision tree construction algorithms, and we outline the distinguishing features of a few of them. See the bibliographical notes for details. With very large data sets, partitioning may be expensive, since it involves repeated copying. Several algorithms have therefore been developed to minimize the I/O and computation cost when the training data are larger than available memory.

Several of the algorithms also prune subtrees of the generated decision tree to reduce **overfitting**: A subtree is overfitted if it has been so highly tuned to the specifics of the training data that it makes many classification errors on other data. A subtree is pruned by replacing it with a leaf node. There are different pruning heuristics; one heuristic uses part of the training data to build the tree and another part of the training data to test it. The heuristic prunes a subtree if it finds that misclassification on the test instances would be reduced if the subtree were replaced by a leaf node.

We can generate classification rules from a decision tree, if we so desire. For each leaf we generate a rule as follows: The left-hand side is the conjunction of all the split conditions on the path to the leaf, and the class is the class of the majority of the training instances at the leaf. An example of such a classification rule is

$$degree = masters \textbf{ and } income > 75,000 \;\Rightarrow\; excellent$$

22.3.2.2 Other Types of Classifiers

There are several types of classifiers other than decision tree classifiers. Two types that have been quite useful are *neural net classifiers* and *Bayesian classifiers*. Neural net classifiers use the training data to train artificial neural nets. There is a large body of literature on neural nets, and we do not consider them further here.

Bayesian classifiers find the distribution of attribute values for each class in the training data; when given a new instance d, they use the distribution information to estimate, for each class c_j, the probability that instance d belongs to class c_j, denoted by $p(c_j|d)$, in a manner outlined here. The class with maximum probability becomes the predicted class for instance d.

To find the probability $p(c_j|d)$ of instance d being in class c_j, Bayesian classifiers use **Bayes' theorem**, which says

$$p(c_j|d) = \frac{p(d|c_j)p(c_j)}{p(d)}$$

where $p(d|c_j)$ is the probability of generating instance d given class c_j, $p(c_j)$ is the probability of occurrence of class c_j, and $p(d)$ is the probability of instance d occurring. Of these, $p(d)$ can be ignored since it is the same for all classes. $p(c_j)$ is simply the fraction of training instances that belong to class c_j.

Finding $p(d|c_j)$ exactly is difficult, since it requires a complete distribution of instances of c_j. To simplify the task, **naive Bayesian classifiers** assume attributes have independent distributions, and thereby estimate

$$p(d|c_j) = p(d_1|c_j) * p(d_2|c_j) * \ldots * p(d_n|c_j)$$

That is, the probability of the instance d occurring is the product of the probability of occurrence of each of the attribute values d_i of d, given the class is c_j.

The probabilities $p(d_i|c_j)$ derive from the distribution of values for each attribute i, for each class class c_j. This distribution is computed from the training instances that belong to each class c_j; the distribution is usually approximated by a histogram. For instance, we may divide the range of values of attribute i into equal intervals, and store the fraction of instances of class c_j that fall in each interval. Given a value d_i for attribute i, the value of $p(d_i|c_j)$ is simply the fraction of instances belonging to class c_j that fall in the interval to which d_i belongs.

A significant benefit of Bayesian classifiers is that they can classify instances with unknown and null attribute values—unknown or null attributes are just omitted from the probability computation. In contrast, decision tree classifiers cannot meaningfully handle situations where an instance to be classified has a null value for a partitioning attribute used to traverse further down the decision tree.

22.3.2.3 Regression

Regression deals with the prediction of a value, rather than a class. Given values for a set of variables, $X_1, X_2, \ldots, X_n$, we wish to predict the value of a variable Y. For instance, we could treat the level of education as a number and income as another number, and, on the basis of these two variables, we wish to predict the likelihood of default, which could be a percentage chance of defaulting, or the amount involved in the default.

One way is to infer coefficients $a_0, a_1, a_1, \ldots, a_n$ such that

$$Y = a_0 + a_1 * X_1 + a_2 * X_2 + \cdots + a_n * X_n$$

Finding such a linear polynomial is called **linear regression**. In general, we wish to find a curve (defined by a polynomial or other formula) that fits the data; the process is also called **curve fitting**.

The fit may only be approximate, because of noise in the data or because the relationship is not exactly a polynomial, so regression aims to find coefficients that give the best possible fit. There are standard techniques in statistics for finding regression coefficients. We do not discuss these techniques here, but the bibliographical notes provide references.

22.3.3 Association Rules

Retail shops are often interested in **associations** between different items that people buy. Examples of such associations are:

- Someone who buys bread is quite likely also to buy milk
- A person who bought the book *Database System Concepts* is quite likely also to buy the book *Operating System Concepts*.

Association information can be used in several ways. When a customer buys a particular book, an online shop may suggest associated books. A grocery shop may decide to place bread close to milk, since they are often bought together, to help shoppers finish their task faster. Or the shop may place them at opposite ends of a row, and place other associated items in between to tempt people to buy those items as well, as the shoppers walk from one end of the row to the other. A shop that offers discounts on one associated item may not offer a discount on the other, since the customer will probably buy the other anyway.

Association Rules

An example of an association rule is

$$bread \Rightarrow milk$$

In the context of grocery-store purchases, the rule says that customers who buy bread also tend to buy milk with a high probability. An association rule must have an associated **population**: the population consists of a set of **instances**. In the grocery-store example, the population may consist of all grocery store purchases; each purchase is an instance. In the case of a bookstore, the population may consist of all people who made purchases, regardless of when they made a purchase. Each customer is an instance. Here, the analyst has decided that when a purchase is made is not significant, whereas for the grocery-store example, the analyst may have decided to concentrate on single purchases, ignoring multiple visits by the same customer.

Rules have an associated *support*, as well as an associated *confidence*. These are defined in the context of the population:

- **Support** is a measure of what fraction of the population satisfies both the antecedent and the consequent of the rule.

  For instance, suppose only 0.001 percent of all purchases include milk and screwdrivers. The support for the rule

  $$milk \Rightarrow screwdrivers$$

  is low. The rule may not even be statistically significant—perhaps there was only a single purchase that included both milk and screwdrivers. Businesses are usually not interested in rules that have low support, since they involve few customers, and are not worth bothering about.

  On the other hand, if 50 percent of all purchases involve milk and bread, then support for rules involving bread and milk (and no other item) is relatively high, and such rules may be worth attention. Exactly what minimum degree of support is considered desirable depends on the application.

- **Confidence** is a measure of how often the consequent is true when the antecedent is true. For instance, the rule

  $$bread \Rightarrow milk$$

  has a confidence of 80 percent if 80 percent of the purchases that include bread also include milk. A rule with a low confidence is not meaningful. In business applications, rules usually have confidences significantly less than 100 percent, whereas in other domains, such as in physics, rules may have high confidences.

  Note that the confidence of $bread \Rightarrow milk$ may be very different from the confidence of $milk \Rightarrow bread$, although both have the same support.

Finding Association Rules

To discover association rules of the form

$$i_1, i_2, \ldots, i_n \Rightarrow i_0$$

we first find sets of items with sufficient support, called **large itemsets**. In our example we find sets of items that are included in a sufficiently large number of instances. We will shortly see how to compute large itemsets.

For each large itemset, we then output all rules with sufficient confidence that involve all and only the elements of the set. For each large itemset S, we output a rule $S - s \Rightarrow s$ for every subset $s \subset S$, provided $S - s \Rightarrow s$ has sufficient confidence; the confidence of the rule is given by support of s divided by support of S.

We now consider how to generate all large itemsets. If the number of possible sets of items is small, a single pass over the data suffices to detect the level of support for all the sets. A count, initialized to 0, is maintained for each set of items. When a purchase record is fetched, the count is incremented for each set of items such that all items in the set are contained in the purchase. For instance, if a purchase included items a, b, and c, counts would be incremented for $\{a\}$, $\{b\}$, $\{c\}$, $\{a, b\}$, $\{b, c\}$, $\{a, c\}$, and $\{a, b, c\}$. Those sets with a sufficiently high count at the end of the pass correspond to items that have a high degree of association.

The number of sets grows exponentially, making the procedure just described infeasible if the number of items is large. Luckily, almost all the sets would normally have very low support; optimizations have been developed to eliminate most such sets from consideration. These techniques use multiple passes on the database, considering only some sets in each pass.

In the **a priori** technique for generating large itemsets, only sets with single items are considered in the first pass. In the second pass, sets with two items are considered, and so on.

At the end of a pass all sets with sufficient support are output as large itemsets. Sets found to have too little support at the end of a pass are eliminated. Once a set is eliminated, none of its supersets needs to be considered. In other words, in pass i we need to count only supports for sets of size i such that all subsets of the set have been found to have sufficiently high support; it suffices to test all subsets of size $i - 1$ to ensure this property. At the end of some pass i, we would find that no set of size i has sufficient support, so we do not need to consider any set of size $i + 1$. Computation then terminates.

22.3.4 Other Types of Associations

Using plain association rules has several shortcomings. One of the major shortcomings is that many associations are not very interesting, since they can be predicted. For instance, if many people buy cereal and many people buy bread, we can predict that a fairly large number of people would buy both, even if there is no connection between the two purchases. What would be interesting is a **deviation** from the expected co-occurrence of the two. In statistical terms, we look for **correlations** between items; correlations can be positive, in that the co-occurrence is higher than would have been expected, or negative, in that the items co-occur less frequently than predicted. See a standard textbook on statistics for more information about correlations.

Another important class of data-mining applications is sequence associations (or correlations). Time-series data, such as stock prices on a sequence of days, form an example of sequence data. Stock-market analysts want to find associations among stock-market price sequences. An example of such a association is the following rule: "Whenever bond rates go up, the stock prices go down within 2 days." Discovering such association between sequences can help us to make intelligent investment decisions. See the bibliographical notes for references to research on this topic.

Deviations from temporal patterns are often interesting. For instance, if a company has been growing at a steady rate each year, a deviation from the usual growth rate is surprising. If sales of winter clothes go down in summer, it is not surprising, since we can predict it from past years; a deviation that we could not have predicted from past experience would be considered interesting. Mining techniques can find deviations from what one would have expected on the basis of past temporal/sequential patterns. See the bibliographical notes for references to research on this topic.

22.3.5 Clustering

Intuitively, clustering refers to the problem of finding clusters of points in the given data. The problem of **clustering** can be formalized from distance metrics in several ways. One way is to phrase it as the problem of grouping points into k sets (for a given k) so that the average distance of points from the *centroid* of their assigned cluster is minimized.[5] Another way is to group points so that the average distance between every pair of points in each cluster is minimized. There are other definitions too; see the bibliographical notes for details. But the intuition behind all these definitions is to group similar points together in a single set.

Another type of clustering appears in classification systems in biology. (Such classification systems do not attempt to *predict* classes, rather they attempt to cluster related items together.) For instance, leopards and humans are clustered under the class mammalia, while crocodiles and snakes are clustered under reptilia. Both mammalia and reptilia come under the common class chordata. The clustering of mammalia has further subclusters, such as carnivora and primates. We thus have **hierarchical clustering**. Given characteristics of different species, biologists have created a complex hierarchical clustering scheme grouping related species together at different levels of the hierarchy.

Hierarchical clustering is also useful in other domains—for clustering documents, for example. Internet directory systems (such as Yahoo's) cluster related documents in a hierarchical fashion (see Section 22.5.5). Hierarchical clustering algorithms can be classified as **agglomerative clustering** algorithms, which start by building small clusters and then creater higher levels, or **divisive clustering** algorithms, which first create higher levels of the hierarchical clustering, then refine each resulting cluster into lower level clusters.

The statistics community has studied clustering extensively. Database research has provided scalable clustering algorithms that can cluster very large data sets (that may not fit in memory). The Birch clustering algorithm is one such algorithm. Intuitively, data points are inserted into a multidimensional tree structure (based on R-trees, described in Section 23.3.5.3), and guided to appropriate leaf nodes based on nearness to representative points in the internal nodes of the tree. Nearby points are thus clustered together in leaf nodes, and summarized if there are more points than fit in memory. Some postprocessing after insertion of all points gives the desired overall clustering. See the bibliographical notes for references to the Birch algorithm, and other techniques for clustering, including algorithms for hierarchical clustering.

An interesting application of clustering is to predict what new movies (or books, or music) a person is likely to be interested in, on the basis of:

1. The person's past preferences in movies
2. Other people with similar past preferences
3. The preferences of such people for new movies

5. The centroid of a set of points is defined as a point whose coordinate on each dimension is the average of the coordinates of all the points of that set on that dimension. For example in two dimensions, the centroid of a set of points $\{ (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \}$ is given by $(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n})$

One approach to this problem is as follows. To find people with similar past preferences we create clusters of people based on their preferences for movies. The accuracy of clustering can be improved by previously clustering movies by their similarity, so even if people have not seen the same movies, if they have seen similar movies they would be clustered together. We can repeat the clustering, alternately clustering people, then movies, then people, and so on till we reache an equilibrium. Given a new user, we find a cluster of users most similar to that user, on the basis of the user's preferences for movies already seen. We then predict movies in movie clusters that are popular with that user's cluster as likely to be interesting to the new user. In fact, this problem is an instance of *collaborative filtering*, where users collaborate in the task of filtering information to find information of interest.

22.3.6 Other Types of Mining

Text mining applies data mining techniques to textual documents. For instance, there are tools that form clusters on pages that a user has visited; this helps users when they browse the history of their browsing to find pages they have visited earlier. The distance between pages can be based, for instance, on common words in the pages (see Section 22.5.1.3). Another application is to classify pages into a Web directory automatically, according to their similarity with other pages (see Section 22.5.5).

Data-visualization systems help users to examine large volumes of data, and to detect patterns visually. Visual displays of data—such as maps, charts, and other graphical representations—allow data to be presented compactly to users. A single graphical screen can encode as much information as a far larger number of text screens. For example, if the user wants to find out whether production problems at plants are correlated to the locations of the plants, the problem locations can be encoded in a special color—say, red—on a map. The user can then quickly discover locations where problems are occurring. The user may then form hypotheses about why problems are occurring in those locations, and may verify the hypotheses quantitatively against the database.

As another example, information about values can be encoded as a color, and can be displayed with as little as one pixel of screen area. To detect associations between pairs of items, we can use a two-dimensional pixel matrix, with each row and each column representing an item. The percentage of transactions that buy both items can be encoded by the color intensity of the pixel. Items with high association will show up as bright pixels in the screen—easy to detect against the darker background.

Data visualization systems do not automatically detect patterns, but provide system support for users to detect patterns. Since humans are very good at detecting visual patterns, data visualization is an important component of data mining.

22.4 Data Warehousing

Large companies have presences in many places, each of which may generate a large volume of data. For instance, large retail chains have hundreds or thousands of stores, whereas insurance companies may have data from thousands of local branches. Further, large organizations have a complex internal organization structure, and there-

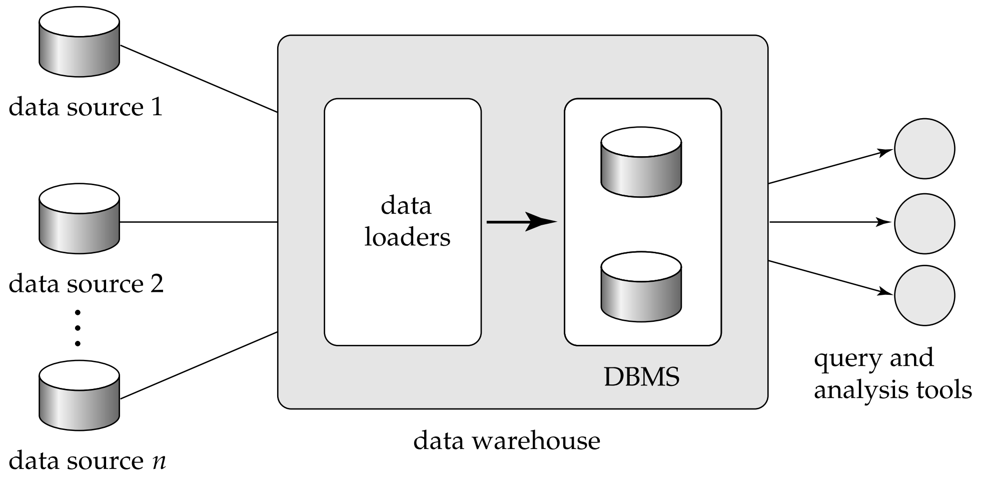


Figure 22.8 Data-warehouse architecture.

fore different data may be present in different locations, or on different operational systems, or under different schemas. For instance, manufacturing-problem data and customer-complaint data may be stored on different database systems. Corporate decision makers require access to information from all such sources. Setting up queries on individual sources is both cumbersome and inefficient. Moreover, the sources of data may store only current data, whereas decision makers may need access to past data as well; for instance, information about how purchase patterns have changed in the past year could be of great importance. Data warehouses provide a solution to these problems.

A **data warehouse** is a repository (or archive) of information gathered from multiple sources, stored under a unified schema, at a single site. Once gathered, the data are stored for a long time, permitting access to historical data. Thus, data warehouses provide the user a single consolidated interface to data, making decision-support queries easier to write. Moreover, by accessing information for decision support from a data warehouse, the decision maker ensures that online transaction-processing systems are not affected by the decision-support workload.

22.4.1 Components of a Data Warehouse

Figure 22.8 shows the architecture of a typical data warehouse, and illustrates the gathering of data, the storage of data, and the querying and data-analysis support. Among the issues to be addressed in building a warehouse are the following:

- **When and how to gather data.** In a **source-driven architecture** for gathering data, the data sources transmit new information, either continually (as transaction processing takes place), or periodically (nightly, for example). In a **destination-driven architecture**, the data warehouse periodically sends requests for new data to the sources.

Unless updates at the sources are replicated at the warehouse via two-phase commit, the warehouse will never be quite up to date with the sources. Two-phase commit is usually far too expensive to be an option, so data warehouses typically have slightly out-of-date data. That, however, is usually not a problem for decision-support systems.

- **What schema to use.** Data sources that have been constructed independently are likely to have different schemas. In fact, they may even use different data models. Part of the task of a warehouse is to perform schema integration, and to convert data to the integrated schema before they are stored. As a result, the data stored in the warehouse are not just a copy of the data at the sources. Instead, they can be thought of as a materialized view of the data at the sources.
- **Data cleansing.** The task of correcting and preprocessing data is called **data cleansing**. Data sources often deliver data with numerous minor inconsistencies, that can be corrected. For example, names are often misspelled, and addresses may have street/area/city names misspelled, or zip codes entered incorrectly. These can be corrected to a reasonable extent by consulting a database of street names and zip codes in each city. Address lists collected from multiple sources may have duplicates that need to be eliminated in a **merge–purge operation**. Records for multiple individuals in a house may be grouped together so only one mailing is sent to each house; this operation is called **householding**.
- **How to propagate updates**. Updates on relations at the data sources must be propagated to the data warehouse. If the relations at the data warehouse are exactly the same as those at the data source, the propagation is straightforward. If they are not, the problem of propagating updates is basically the *view-maintenance* problem, which was discussed in Section 14.5.
- **What data to summarize.** The raw data generated by a transaction-processing system may be too large to store online. However, we can answer many queries by maintaining just summary data obtained by aggregation on a relation, rather than maintaining the entire relation. For example, instead of storing data about every sale of clothing, we can store total sales of clothing by item-name and category.

  Suppose that a relation r has been replaced by a summary relation s. Users may still be permitted to pose queries as though the relation r were available online. If the query requires only summary data, it may be possible to transform it into an equivalent one using s instead; see Section 14.5.

22.4.2 Warehouse Schemas

Data warehouses typically have schemas that are designed for data analysis, using tools such as OLAP tools. Thus, the data are usually multidimensional data, with dimension attributes and measure attributes. Tables containing multidimensional data are called **fact tables** and are usually very large. A table recording sales information for a retail store, with one tuple for each item that is sold, is a typical example of a fact table. The dimensions of the *sales* table would include what the item is (usually an item identifier such as that used in bar codes), the date when the item is sold, which location (store) the item was sold from, which customer bought the item, and so on. The measure attributes may include the number of items sold and the price of the items.

To minimize storage requirements, dimension attributes are usually short identifiers that are foreign keys into other other tables called **dimension tables**. For instance, a fact table *sales* would have attributes *item-id*, *store-id*, *customer-id*, and *date*, and measure attributes *number* and *price*. The attribute *store-id* is a foreign key into a dimension table *store*, which has other attributes such as store location (city, state, country). The *item-id* attribute of the *sales* table would be a foreign key into a dimension table *item-info*, which would contain information such as the name of the item, the category to which the item belongs, and other item details such as color and size. The *customer-id* attribute would be a foreign key into a *customer* table containing attributes such as name and address of the customer. We can also view the *date* attribute as a foreign key into a *date-info* table giving the month, quarter, and year of each date.

The resultant schema appears in Figure 22.9. Such a schema, with a fact table, multiple dimension tables, and foreign keys from the fact table to the dimension tables, is called a **star schema**. More complex data warehouse designs may have multiple levels of dimension tables; for instance, the *item-info* table may have an attribute *manufacturer-id* that is a foreign key into another table giving details of the manufacturer. Such schemas are called *snowflake schema*s. Complex data warehouse designs may also have more than one fact table.

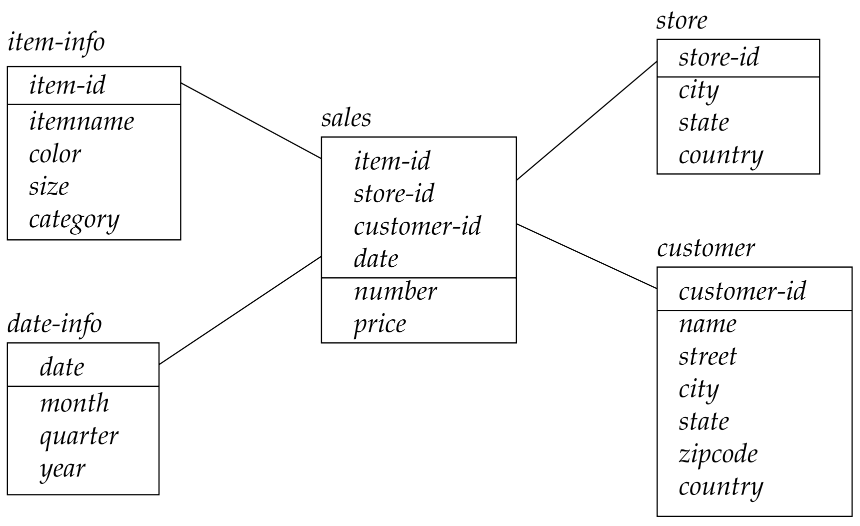


Figure 22.9 Star schema for a data warehouse.

22.5 Information-Retrieval Systems

The field of **information retrieval** has developed in parallel with the field of databases. In the traditional model used in the field of information retrieval, information is organized into documents, and it is assumed that there is a large number of documents. Data contained in documents is unstructured, without any associated schema. The process of information retrieval consists of locating relevant documents, on the basis of user input, such as keywords or example documents.

The Web provides a convenient way to get to, and to interact with, information sources across the Internet. However, a persistent problem facing the Web is the explosion of stored information, with little guidance to help the user to locate what is interesting. Information retrieval has played a critical role in making the Web a productive and useful tool, especially for researchers.

Traditional examples of information-retrieval systems are online library catalogs and online document-management systems such as those that store newspaper articles. The data in such systems are organized as a collection of *documents*; a newspaper article or a catalog entry (in a library catalog) are examples of documents. In the context of the Web, usually each HTML page is considered to be a document.

A user of such a system may want to retrieve a particular document or a particular class of documents. The intended documents are typically described by a set of **keywords**—for example, the keywords "database system" may be used to locate books on database systems, and the keywords "stock" and "scandal" may be used to locate articles about stock-market scandals. Documents have associated with them a set of keywords, and documents whose keywords contain those supplied by the user are retrieved.

Keyword-based information retrieval can be used not only for retrieving textual data, but also for retrieving other types of data, such as video or audio data, that have descriptive keywords associated with them. For instance, a video movie may have associated with it keywords such as its title, director, actors, type, and so on.

There are several differences between this model and the models used in traditional database systems.

- Database systems deal with several operations that are not addressed in information-retrieval systems. For instance, database systems deal with updates and with the associated transactional requirements of concurrency control and durability. These matters are viewed as less important in information systems. Similarly, database systems deal with structured information organized with relatively complex data models (such as the relational model or object-oriented data models), whereas information-retrieval systems traditionally have used a much simpler model, where the information in the database is organized simply as a collection of unstructured documents.

- Information-retrieval systems deal with several issues that have not been addressed adequately in database systems. For instance, the field of information retrieval has dealt with the problems of managing unstructured documents, such as approximate searching by keywords, and of ranking of documents on estimated degree of relevance of the documents to the query.

22.5.1 Keyword Search

Information-retrieval systems typically allow query expressions formed using keywords and the logical connectives *and, or*, and *not*. For example, a user could ask for all documents that contain the keywords "motorcycle *and* maintenance," or documents that contain the keywords "computer *or* microprocessor," or even documents that contain the keyword "computer *but not* database." A query containing keywords without any of the above connectives is assumed to have *and*s implicitly connecting the keywords.

In **full text** retrieval, all the words in each document are considered to be keywords. For unstructured documents, full text retrieval is essential since there may be no information about what words in the document are keywords. We shall use the word **term** to refer to the words in a document, since all words are keywords.

In its simplest form an information retrieval system locates and returns all documents that contain all the keywords in the query, if the query has no connectives; connectives are handled as you would expect. More sophisticated systems estimate relevance of documents to a query so that the documents can be shown in order of estimated relevance. They use information about term occurrences, as well as hyperlink information, to estimate relevance; Section 22.5.1.1 and 22.5.1.2 outline how to do so. Section 22.5.1.3 outlines how to define similarity of documents, and use similarity for searching. Some systems also attempt to provide a better set of answers by using the meanings of terms, rather than just the syntactic occurrence of terms, as outlined in Section 22.5.1.4.

22.5.1.1 Relevance Ranking Using Terms

The set of all documents that satisfy a query expression may be very large; in particular, there are billions of documents on the Web, and most keyword queries on a Web search engine find hundreds of thousands of documents containing the keywords. Full text retrieval makes this problem worse: Each document may contain many terms, and even terms that are only mentioned in passing are treated equivalently with documents where the term is indeed relevant. Irrelevant documents may get retrieved as a result.

Information retrieval systems therefore estimate relevance of documents to a query, and return only highly ranked documents as answers. Relevance ranking is not an exact science, but there are some well-accepted approaches.

The first question to address is, given a particular term t, how relevant is a particular document d to the term. One approach is to use the the number of occurrences of the term in the document as a measure of its relevance, on the assumption that relevant terms are likely to be mentioned many times in a document. Just counting the number of occurrences of a term is usually not a good indicator: First, the number of occurrences depends on the length of the document, and second, a document containing 10 occurrences of a term may not be 10 times as relevant as a document containing one occurrence.

One way of measuring $r(d, t)$, the relevance of a document d to a term t, is

$$r(d, t) = \log\left(1 + \frac{n(d, t)}{n(d)}\right)$$

where $n(d)$ denotes the number of terms in the document and $n(d, t)$ denotes the number of occurrences of term t in the document d. Observe that this metric takes the length of the document into account. The relevance grows with more occurrences of a term in the document, although it is not directly proportional to the number of occurrences.

Many systems refine the above metric by using other information. For instance, if the term occurs in the title, or the author list, or the abstract, the document would be considered more relevant to the term. Similarly, if the first occurrence of a term is late in the document, the document may be considered less relevant than if the first occurrence is early in the document. The above notions can be formalized by extensions of the formula we have shown for $r(d, t)$. In the information retrieval community, the relevance of a document to a term is referred to as **term frequency**, regardless of the exact formula used.

A query Q may contain multiple keywords. The relevance of a document to a query with two or more keywords is estimated by combining the relevance measures of the document to each keyword. A simple way of combining the measures is to add them up. However, not all terms used as keywords are equal. Suppose a query uses two terms, one of which occurs frequently, such as "web," and another that is less frequent, such as "Silberschatz." A document containing "Silberschatz" but not "web" should be ranked higher than a document containing the term "web" but not "Silberschatz."

To fix the above problem, weights are assigned to terms using the **inverse document frequency**, defined as $1/n(t)$, where $n(t)$ denotes the number of documents (among those indexed by the system) that contain the term t. The **relevance** of a document d to a set of terms Q is then defined as

$$r(d, Q) = \sum_{t \in Q} \frac{r(d, t)}{n(t)}$$

This measure can be further refined if the user is permitted to specify weights $w(t)$ for terms in the query, in which case the user-specified weights are also taken into account by using $w(t)/n(t)$ in place of $1/n(t)$.

Almost all text documents (in English) contain words such as "and," "or," "a," and so on, and hence these words are useless for querying purposes since their inverse document frequency is extremely low. Information-retrieval systems define a set of words, called **stop words**, containing 100 or so of the most common words, and remove this set from the document when indexing; such words are not used as keywords, and are discarded if present in the keywords supplied by the user.

Another factor taken into account when a query contains multiple terms is the **proximity** of the term in the document. If the terms occur close to each other in the document, the document would be ranked higher than if they occur far apart. The formula for $r(d, Q)$ can be modified to take proximity into account.

Given a query Q, the job of an information retrieval system is to return documents in descending order of their relevance to Q. Since there may be a very large number of documents that are relevant, information retrieval systems typically return only the first few documents with the highest degree of estimated relevance, and permit users to interactively request further documents.

22.5.1.2 Relevance Using Hyperlinks

Early Web search engines ranked documents by using only relevance measures similar to those described in Section 22.5.1.1. However, researchers soon realized that Web documents have information that plain text documents do not have, namely hyperlinks. And in fact, the relevance ranking of a document is affected more by hyperlinks that point *to* the document, than by hyperlinks going out of the document.

The basic idea of site ranking is to find sites that are popular, and to rank pages from such sites higher than pages from other sites. A site is identified by the internet address part of the URL, such as www.bell-labs.com in a URL http://www.bell-labs.com/topic/books/db-book. A site usually contains multiple Web pages. Since most searches are intended to find information from popular sites, ranking pages from popular sites higher is generally a good idea. For instance, the term "google" may occur in vast numbers of pages, but the site google.com is the most popular among the sites with pages that contain the term "google". Documents from google.com containing the term "google" would therefore be ranked as the most relevant to the term "google".

This raises the question of how to define the popularity of a site. One way would be to find how many times a site is accessed. However, getting such information is impossible without the cooperation of the site, and is infeasible for a Web search engine to implement. A very effective alternative uses hyperlinks; it defines $p(s)$, the **popularity of a site** s, as the number of sites that contain at least one page with a link to site s.

Traditional measures of relevance of the page (which we saw in Section 22.5.1.2) can be combined with the popularity of the site containing the page to get an overall measure of the relevance of the page. Pages with high overall relevance value are returned as answers to a query, as before.

Note also that we used the popularity of a *site* as a measure of relevance of individual pages at the site, not the popularity of individual *pages*. There are at least two reasons for this. First, most sites contain only links to root pages of other sites, so all other pages would appear to have almost zero popularity, when in fact they may be accessed quite frequently by following links from the root page. Second, there are far fewer sites than pages, so computing and using popularity of sites is cheaper than computing and using popularity of pages.

There are more refined notions of popularity of sites. For instance, a link from a popular site to another site s may be considered to be a better indication of the popularity of s than a link to s from a less popular site.[6] This notion of popularity

6. This is similar in some sense to giving extra weight to endorsements of products by celebrities (such as film stars), so its significance is open to question!

is in fact circular, since the popularity of a site is defined by the popularity of other sites, and there may be cycles of links between sites. However, the popularity of sites can be defined by a system of simultaneous linear equations, which can be solved by matrix manipulation techniques. The linear equations are defined in such a way that they have a unique and well-defined solution.

The popular Web search engine google.com uses the referring-site popularity idea in its definition **page rank**, which is a measure of popularity of a page. This approach of ranking of pages gave results so much better than previously used ranking techniques, that google.com became a widely used search engine, in a rather short period of time.

There is another, somewhat similar, approach, derived interestingly from a theory of social networking developed by sociologists in the 1950s. In the social networking context, the goal was to define the prestige of people. For example, the president of the United States has high prestige since a large number of people know him. If someone is known by multiple prestigious people, then she also has high prestige, even if she is not known by as large a number of people.

The above idea was developed into a notion of *hubs* and *authorities* that takes into account the presence of directories that link to pages containing useful information. A **hub** is a page that stores links to many pages; it does not in itself contain actual information on a topic, but points to pages that contain actual information. In contrast, an **authority** is a page that contains actual information on a topic, although it may not be directly pointed to by many pages. Each page then gets a prestige value as a hub (*hub-prestige*), and another prestige value as an authority (*authority-prestige*). The definitions of prestige, as before, are cyclic and are defined by a set of simultaneous linear equations. A page gets higher hub-prestige if it points to many pages with high authority-prestige, while a page gets higher authority-prestige if it is pointed to by many pages with high hub-prestige. Given a query, pages with highest authority-prestige are ranked higher than other pages. See the bibliographical notes for references giving further details.

22.5.1.3 Similarity-Based Retrieval

Certain information-retrieval systems permit **similarity-based retrieval**. Here, the user can give the system document A, and ask the system to retrieve documents that are "similar" to A. The similarity of a document to another may be defined, for example, on the basis of common terms. One approach is to find k terms in A with highest values of $r(d, t)$, and to use these k terms as a query to find relevance of other documents. The terms in the query are themselves weighted by $r(d, t)$.

If the set of documents similar to A is large, the system may present the user a few of the similar documents, allow him to choose the most relevant few, and start a new search based on similarity to A *and* to the chosen documents. The resultant set of documents is likely to be what the user intended to find.

The same idea is also used to help users who find many documents that appear to be relevant on the basis of the keywords, but are not. In such a situation, instead of adding further keywords to the query, users may be allowed to identify one or a few of the returned documents as relevant; the system then uses the identified documents to find other similar ones. The resultant set of documents is likely to be what the user intended to find.

22.5.1.4 Synonyms and Homonyms

Consider the problem of locating documents about motorcycle maintenance for the keywords “motorcycle” and “maintenance.” Suppose that the keywords for each document are the words in the title and the names of the authors. The document titled *Motorcycle Repair* would not be retrieved, since the word “maintenance” does not occur in its title.

We can solve that problem by making use of **synonyms**. Each word can have a set of synonyms defined, and the occurrence of a word can be replaced by the *or* of all its synonyms (including the word itself). Thus, the query “motorcycle *and* repair” can be replaced by “motorcycle *and* (repair *or* maintenance).” This query would find the desired document.

Keyword-based queries also suffer from the opposite problem, of **homonyms**, that is single words with multiple meanings. For instance, the word object has different meanings as a noun and as a verb. The word table may refer to a dinner table, or to a relational table. Some keyword query systems attempt to disambiguate the meaning of words in documents, and when a user poses a query, they find out the intended meaning by asking the user. The returned documents are those that use the term in the intended meaning of the user. However, disambiguating meanings of words in documents is not an easy task, so not many systems implement this idea.

In fact, a danger even with using synonyms to extend queries is that the synonyms may themselves have different meanings. Documents that use the synonyms with an alternative intended meaning would be retrieved. The user is then left wondering why the system thought that a particular retrieved document is relevant, if it contains neither the keywords the user specified, nor words whose intended meaning in the document is synonymous with specified keywords! It is therefore advisable to verify synonyms with the user, before using them to extend a query submitted by the user.

22.5.2 Indexing of Documents

An effective index structure is important for efficient processing of queries in an information-retrieval system. Documents that contain a specified keyword can be efficiently located by using an **inverted index**, which maps each keyword K_i to the set S_i of (identifiers of) the documents that contain K_i. To support relevance ranking based on proximity of keywords, such an index may provide not just identifiers of documents, but also a list of locations in the document where the keyword appears. Since such indices must be stored on disk, the index organization also attempts to minimize the number of I/O operations to retrieve the set of (identifiers of) documents that contain a keyword. Thus, the system may attempt to keep the set of documents for a keyword in consecutive disk pages.

The *and* operation finds documents that contain all of a specified set of keywords $K_1, K_2, \ldots, K_n$. We implement the *and* operation by first retrieving the sets of document identifiers $S_1, S_2, \ldots, S_n$ of all documents that contain the respective keywords.

The intersection, $S_1 \cap S_2 \cap \cdots \cap S_n$, of the sets gives the document identifiers of the desired set of documents. The *or* operation gives the set of all documents that contain at least one of the keywords $K_1, K_2, \ldots, K_n$. We implement the *or* operation by computing the union, $S_1 \cup S_2 \cup \cdots \cup S_n$, of the sets. The *not* operation finds documents that do not contain a specified keyword K_i. Given a set of document identifiers S, we can eliminate documents that contain the specified keyword K_i by taking the difference $S - S_i$, where S_i is the set of identifiers of documents that contain the keyword K_i.

Given a set of keywords in a query, many information retrieval systems do not insist that the retrieved documents contain all the keywords (unless an *and* operation is explicitly used). In this case, all documents containing at least one of the words are retrieved (as in the *or* operation), but are ranked by their relevance measure.

To use term frequency for ranking, the index structure should additionally maintain the number of times terms occur in each document. To reduce this effort, they may use a compressed representation with only a few bits, which approximates the term frequency. The index should also store the document frequency of each term (that is, the number of documents in which the term appears).

22.5.3 Measuring Retrieval Effectiveness

Each keyword may be contained in a large number of documents; hence, a compact representation is critical to keep space usage of the index low. Thus, the sets of documents for a keyword are maintained in a compressed form. So that storage space is saved, the index is sometimes stored such that the retrieval is approximate; a few relevant documents may not be retrieved (called a **false drop** or **false negative**), or a few irrelevant documents may be retrieved (called a **false positive**). A good index structure will not have *any* false drops, but may permit a few false positives; the system can filter them away later by looking at the keywords that they actually contain. In Web indexing, false positives are not desirable either, since the actual document may not be quickly accessible for filtering.

Two metrics are used to measure how well an information-retrieval system is able to answer queries. The first, **precision**, measures what percentage of the retrieved documents are actually relevant to the query. The second, **recall**, measures what percentage of the documents relevant to the query were retrieved. Ideally both should be 100 percent.

Precision and recall are also important measures for understanding how well a particular document ranking strategy performs. Ranking strategies can result in false negatives and false positives, but in a more subtle sense.

- False negatives may occur when documents are ranked, because relevant documents get low rankings; if we fetched all documents down to documents with very low ranking there would be very few false negatives. However, humans would rarely look beyond the first few tens of returned documents, and may thus miss relevant documents because they are not ranked among the top few. Exactly what is a false negative depends on how many documents are examined.

  Therefore instead of having a single number as the measure of recall, we can measure the recall as a function of the number of documents fetched.

- False positives may occur because irrelevant documents get higher rankings than relevant documents. This too depends on how many documents are examined. One option is to measure precision as a function of number of documents fetched.

A better and more intuitive alternative for measuring precision is to measure it as a function of recall. With this combined measure, both precision and recall can be computed as a function of number of documents, if required.

For instance, we can say that with a recall of 50 percent the precision was 75 percent, whereas at a recall of 75 percent the precision dropped to 60 percent. In general, we can draw a graph relating precision to recall. These measures can be computed for individual queries, then averaged out across a suite of queries in a query benchmark.

Yet another problem with measuring precision and recall lies in how to define which documents are really relevant and which are not. In fact, it requires understanding of natural language, and understanding of the intent of the query, to decide if a document is relevant or not. Researchers therefore have created collections of documents and queries, and have manually tagged documents as relevant or irrelevant to the queries. Different ranking systems can be run on these collections to measure their average precision and recall across multiple queries.

22.5.4 Web Search Engines

Web crawlers are programs that locate and gather information on the Web. They recursively follow hyperlinks present in known documents to find other documents. A crawler retrieves the documents and adds information found in the documents to a combined index; the document is generally not stored, although some search engines do cache a copy of the document to give clients faster access to the documents.

Since the number of documents on the Web is very large, it is not possible to crawl the whole Web in a short period of time; and in fact, all search engines cover only some portions of the Web, not all of it, and their crawlers may take weeks or months to perform a single crawl of all the pages they cover. There are usually many processes, running on multiple machines, involved in crawling. A database stores a set of links (or sites) to be crawled; it assigns links from this set to each crawler process. New links found during a crawl are added to the database, and may be crawled later if they are not crawled immediately. Pages found during a crawl are also handed over to an indexing system, which may be running on a different machine. Pages have to be refetched (that is, links recrawled) periodically to obtain updated information, and to discard sites that no longer exist, so that the information in the search index is kept reasonably up to date.

The indexing system itself runs on multiple machines in parallel. It is not a good idea to add pages to the same index that is being used for queries, since doing so would require concurrency control on the index, and affect query and update performance. Instead, one copy of the index is used to answer queries while another copy is updated with newly crawled pages. At periodic intervals the copies switch over, with the old one being updated while the new copy is being used for queries.

To support very high query rates, the indices may be kept in main memory, and there are multiple machines; the system selectively routes queries to the machines to balance the load among them.

22.5.5 Directories

A typical library user may use a catalog to locate a book for which she is looking. When she retrieves the book from the shelf, however, she is likely to *browse* through other books that are located nearby. Libraries organize books in such a way that related books are kept close together. Hence, a book that is physically near the desired book may be of interest as well, making it worthwhile for users to browse through such books.

To keep related books close together, libraries use a **classification hierarchy**. Books on science are classified together. Within this set of books, there is a finer classification, with computer-science books organized together, mathematics books organized together, and so on. Since there is a relation between mathematics and computer science, relevant sets of books are stored close to each other physically. At yet another level in the classification hierarchy, computer-science books are broken down into subareas, such as operating systems, languages, and algorithms. Figure 22.10 illustrates a classification hierarchy that may be used by a library. Because books can be kept at only one place, each book in a library is classified into exactly one spot in the classification hierarchy.

In an information retrieval system, there is no need to store related documents close together. However, such systems need to *organize documents logically* so as to permit browsing. Thus, such a system could use a classification hierarchy similar to

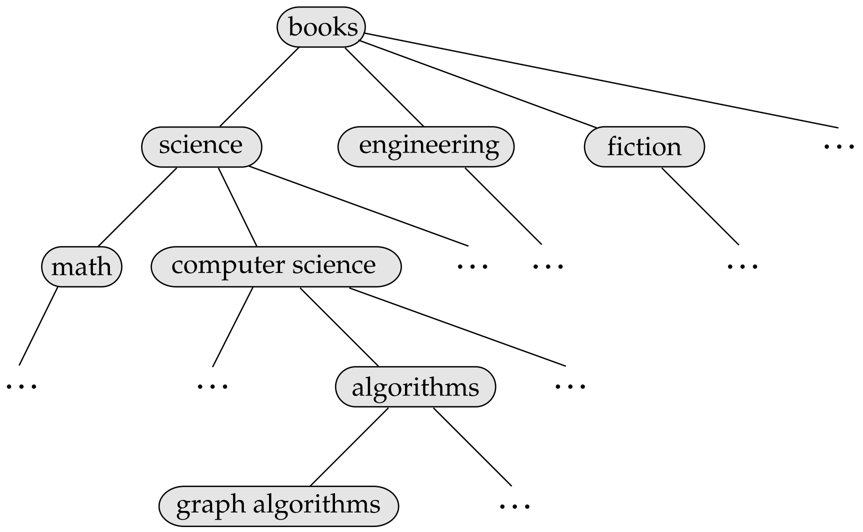


Figure 22.10 A classification hierarchy for a library system.

one that libraries use, and, when it displays a particular document, it can also display a brief description of documents that are close in the hierarchy.

In an information retrieval system, there is no need to keep a document in a single spot in the hierarchy. A document that talks of mathematics for computer scientists could be classified under mathematics as well as under computer science. All that is stored at each spot is an identifier of the document (that is, a pointer to the document), and it is easy to fetch the contents of the document by using the identifier.

As a result of this flexibility, not only can a document be classified under two locations, but also a subarea in the classification hierarchy can itself occur under two areas. The class of "graph algorithm" document can appear both under mathematics and under computer science. Thus, the classification hierarchy is now a directed acyclic graph (DAG), as shown in Figure 22.11. A graph-algorithm document may appear in a single location in the DAG, but can be reached via multiple paths.

A **directory** is simply a classification DAG structure. Each leaf of the directory stores links to documents on the topic represented by the leaf. Internal nodes may also contain links, for example to documents that cannot be classified under any of the child nodes.

To find information on a topic, a user would start at the root of the directory and follow paths down the DAG until reaching a node representing the desired topic. While browsing down the directory, the user can find not only documents on the topic he is interested in, but also find related documents and related classes in the classification hierarchy. The user may learn new information by browsing through documents (or subclasses) within the related classes.

Organizing the enormous amount of information available on the Web into a directory structure is a daunting task.

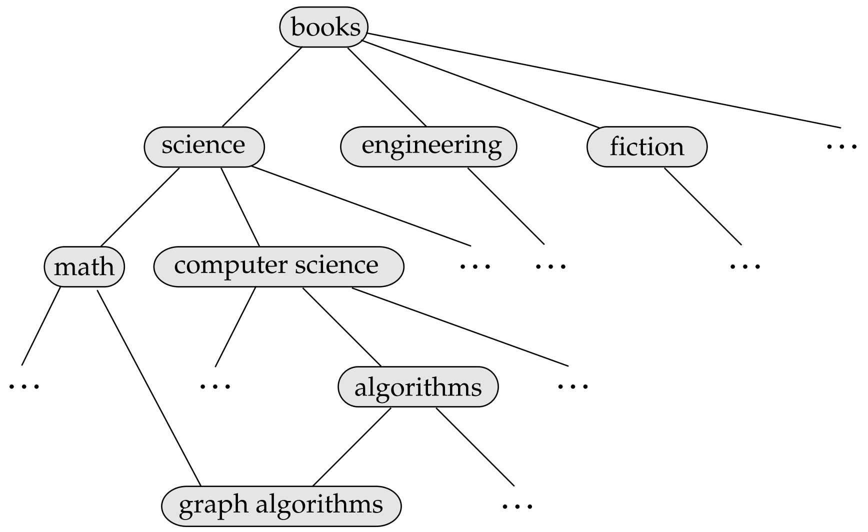


Figure 22.11 A classification DAG for a library information retrieval system.

- The first problem is determining what exactly the directory hierarchy should be.
- The second problem is, given a document, deciding which nodes of the directory are categories relevant to the document.

To tackle the first problem, portals such as Yahoo have teams of "internet librarians" who come up with the classification hierarchy and continually refine it. The *Open Directory Project* is a large collaborative effort, with different volunteers being responsible for organizing different branches of the directory.

The second problem can also be tackled manually by librarians, or Web site maintainers may be responsible for deciding where their sites should lie in the hierarchy. There are also techniques for automatically deciding the location of documents based on computing their similarity to documents that have already been classified.

22.6 Summary

- Decision-support systems analyze online data collected by transaction-processing systems, to help people make business decisions. Since most organizations are extensively computerized today, a very large body of information is available for decision support. Decision-support systems come in various forms, including OLAP systems and data mining systems.
- Online analytical processing (OLAP) tools help analysts view data summarized in different ways, so that they can gain insight into the functioning of an organization.
  - ☐ OLAP tools work on multidimensional data, characterized by dimension attributes and measure attributes.
  - ☐ The data cube consists of multidimensional data summarized in different ways. Precomputing the data cube helps speed up queries on summaries of data.
  - ☐ Cross-tab displays permit users to view two dimensions of multidimensional data at a time, along with summaries of the data.
  - ☐ Drill down, rollup, slicing, and dicing are among the operations that users perform with OLAP tools.
- The OLAP component of the SQL:1999 standard provides a variety of new functionality for data analysis, including new aggregate functions, cube and rollup operations, ranking functions, windowing functions, which support summarization on moving windows, and partitioning, with windowing and ranking applied inside each partition.
- Data mining is the process of semiautomatically analyzing large databases to find useful patterns. There are a number of applications of data mining, such as prediction of values based on past examples, finding of associations between purchases, and automatic clustering of people and movies.

- Classification deals with predicting the class of test instances, by using attributes of the test instances, based on attributes of training instances, and the actual class of training instances. Classification can be used, for instance, to predict credit-worthiness levels of new applicants, or to predict the performance of applicants to a university.
  There are several types of classifiers, such as
  - ☐ Decision-tree classifiers. These perform classification by constructing a tree based on training instances with leaves having class labels. The tree is traversed for each test instance to find a leaf, and the class of the leaf is the predicted class.
    Several techniques are available to construct decision trees, most of them based on greedy heuristics.
  - ☐ Bayesian classifiers are simpler to construct than decision-tree classifiers, and work better in the case of missing/null attribute values.

- Association rules identify items that co-occur frequently, for instance, items that tend to be bought by the same customer. Correlations look for deviations from expected levels of association.

- Other types of data mining include clustering, text mining, and data visualization.

- Data warehouses help gather and archive important operational data. Warehouses are used for decision support and analysis on historical data, for instance to predict trends. Data cleansing from input data sources is often a major task in data warehousing. Warehouse schemas tend to be multidimensional, involving one or a few very large fact tables and several much smaller dimension tables.

- Information retrieval systems are used to store and query textual data such as documents. They use a simpler data model than do database systems, but provide more powerful querying capabilities within the restricted model.
  Queries attempt to locate documents that are of interest by specifying, for example, sets of keywords. The query that a user has in mind usually cannot be stated precisely; hence, information-retrieval systems order answers on the basis of potential relevance.

- Relevance ranking makes use of several types of information, such as:
  - ☐ Term frequency: how important each term is to each document.
  - ☐ Inverse document frequency.
  - ☐ Site popularity. Page rank and hub/authority rank are two ways to assign importance to sites on the basis of links to the site.

- Similarity of documents is used to retrieve documents similar to an example document. Synonyms and homonyms complicate the task of information retrieval.

- Precision and recall are two measures of the effectiveness of an information retrieval system.
- Directory structures are used to classify documents with other similar documents.

Review Terms

- Decision-support systems
- Statistical analysis
- Multidimensional data
  - ☐ Measure attributes
  - ☐ Dimension attributes
- Cross-tabulation
- Data cube
- Online analytical processing (OLAP)
  - ☐ Pivoting
  - ☐ Slicing and dicing
  - ☐ Rollup and drill down
- Multidimensional OLAP (MOLAP)
- Relational OLAP (ROLAP)
- Hybrid OLAP (HOLAP)
- Extended aggregation
  - ☐ Variance
  - ☐ Standard deviation
  - ☐ Correlation
  - ☐ Regression
- Ranking functions
  - ☐ Rank
  - ☐ Dense rank
  - ☐ Partition by
- Windowing
- Data mining
- Prediction
- Associations
- Classification
  - ☐ Training data
  - ☐ Test data
- Decision-tree classifiers
  - ☐ Partitioning attribute
  - ☐ Partitioning condition
  - ☐ Purity
    - — Gini measure
    - — Entropy measure
  - ☐ Information gain
  - ☐ Information content
  - ☐ Information gain ratio
  - ☐ Continuous-valued attribute
  - ☐ Categorical attribute
  - ☐ Binary split
  - ☐ Multiway split
  - ☐ Overfitting
- Bayesian classifiers
  - ☐ Bayes theorem
  - ☐ Naive Bayesian classifiers
- Regression
  - ☐ Linear regression
  - ☐ Curve fitting
- Association rules
  - ☐ Population
  - ☐ Support
  - ☐ Confidence
  - ☐ Large itemsets
- Other types of associations
- Clustering
  - ☐ Hierarchical clustering
  - ☐ Agglomerative clustering
  - ☐ Divisive clustering
- Text mining
- Data visualization
- Data warehousing
  - ☐ Gathering data
  - ☐ Source-driven architecture

  - ☐ Destination-driven architecture
  - ☐ Data cleansing
    - — Merge–purge
    - — Householding
- Warehouse schemas
  - ☐ Fact table
  - ☐ Dimension tables
  - ☐ Star schema
- Information retrieval systems
- Keyword search
- Full text retrieval
- Term
- Relevance ranking
  - ☐ Term frequency
  - ☐ Inverse document frequency
  - ☐ Relevance
  - ☐ Proximity
- Stop words
- Relevance using hyperlinks
  - ☐ Site popularity
  - ☐ Page rank
  - ☐ Hub/authority ranking
- Similarity-based retrieval
- Synonyms
- Homonyms
- Inverted index
- False drop
- False negative
- False positive
- Precision
- Recall
- Web crawlers
- Directories
- Classification hierarchy

Exercises

22.1 For each of the SQL aggregate functions **sum, count, min** and **max**, show how to compute the aggregate value on a multiset $S_1 \cup S_2$, given the aggregate values on multisets S_1 and S_2.

Based on the above, give expressions to compute aggregate values with grouping on a subset S of the attributes of a relation $r(A, B, C, D, E)$, given aggregate values for grouping on attributes $T \supseteq S$, for the following aggregate functions:

a. **sum, count, min** and **max**
b. **avg**
c. standard deviation

22.2 Show how to express **group by cube**(a, b, c, d) using **rollup**; your answer should have only one **group by** clause.

22.3 Give an example of a pair of groupings that cannot be expressed by using a single **group by** clause with **cube** and **rollup**.

22.4 Given a relation $S(student, subject, marks)$, write a query to find the top n students by total marks, by using ranking.

22.5 Given relation $r(a, b, d, d)$, Show how to use the extended SQL features to generate a histogram of d versus a, dividing a into 20 equal-sized partitions (that is, where each partition contains 5 percent of the tuples in r, sorted by a).

22.6 Write a query to find cumulative balances, equivalent to that shown in Section 22.2.5, but without using the extended SQL windowing constructs.

22.7 Consider the *balance* attribute of the *account* relation. Write an SQL query to compute a histogram of *balance* values, dividing the range 0 to the maximum account balance present, into three equal ranges.

22.8 Consider the *sales* relation from Section 22.2. Write an SQL query to compute the cube operation on the relation, giving the relation in Figure 22.2. Do not use the **with cube** construct.

22.9 Construct a decision tree classifier with binary splits at each node, using tuples in relation $r(A, B, C)$ shown below as training data; attribute C denotes the class. Show the final tree, and with each node show the best split for each attribute along with its information gain value.

$$(1, 2, a), (2, 1, a), (2, 5, b), (3, 3, b), (3, 6, b), (4, 5, b), (5, 5, c), (6, 3, b), (6, 7, c)$$

22.10 Suppose there are two classification rules, one that says that people with salaries between \$10,000 and \$20,000 have a credit rating of *good*, and another that says that people with salaries between \$20,000 and \$30,000 have a credit rating of *good*. Under what conditions can the rules be replaced, without any loss of information, by a single rule that says people with salaries between \$10,000 and \$30,000 have a credit rating of *good*.

22.11 Suppose half of all the transactions in a clothes shop purchase jeans, and one third of all transactions in the shop purchase T-shirts. Suppose also that half of the transactions that purchase jeans also purchase T-shirts. Write down all the (nontrivial) association rules you can deduce from the above information, giving support and confidence of each rule.

22.12 Consider the problem of finding large itemsets.

a. Describe how to find the support for a given collection of itemsets by using a single scan of the data. Assume that the itemsets and associated information, such as counts, will fit in memory.

b. Suppose an itemset has support less than j. Show that no superset of this itemset can have support greater than or equal to j.

22.13 Describe benefits and drawbacks of a source-driven architecture for gathering of data at a data-warehouse, as compared to a destination-driven architecture.

22.14 Consider the schema depicted in Figure 22.9. Give an SQL:1999 query to summarize sales numbers and price by store and date, along with the hierarchies on store and date.

22.15 Compute the relevance (using appropriate definitions of term frequency and inverse document frequency) of each of the questions in this chapter to the query “SQL relation.”

22.16 What is the difference between a false positive and a false drop? If it is essential that no relevant information be missed by an information retrieval query, is it acceptable to have either false positives or false drops? Why?

22.17 Suppose you want to find documents that contain at least k of a given set of n keywords. Suppose also you have a keyword index that gives you a (sorted) list of identifiers of documents that contain a specified keyword. Give an efficient algorithm to find the desired set of documents.

Bibliographical Notes

Gray et al. [1995] and Gray et al. [1997] describe the data-cube operator. Efficient algorithms for computing data cubes are described by Agarwal et al. [1996], Harinarayan et al. [1996] and Ross and Srivastava [1997]. Descriptions of extended aggregation support in SQL:1999 can be found in the product manuals of database systems such as Oracle and IBM DB2. Definitions of statistical functions can be found in standard statistics textbooks such as Bulmer [1979] and Ross [1999].

Witten and Frank [1999] and Han and Kamber [2000] provide textbook coverage of data mining. Mitchell [1997] is a classic textbook on machine learning, and covers classification techniques in detail. Fayyad et al. [1995] presents an extensive collection of articles on knowledge discovery and data mining. Kohavi and Provost [2001] presents a collection of articles on applications of data mining to electronic commerce.

Agrawal et al. [1993] provides an early overview of data mining in databases. Algorithms for computing classifiers with large training sets are described by Agrawal et al. [1992] and Shafer et al. [1996]; the decision tree construction algorithm described in this chapter is based on the SPRINT algorithm of Shafer et al. [1996]. Agrawal and Srikant [1994] was an early paper on association rule mining. Algorithms for mining of different forms of association rules are described by Srikant and Agrawal [1996a] and Srikant and Agrawal [1996b]. Chakrabarti et al. [1998] describes techniques for mining surprising temporal patterns.

Clustering has long been studied in the area of statistics, and Jain and Dubes [1988] provides textbook coverage of clustering. Ng and Han [1994] describes spatial clustering techniques. Clustering techniques for large datasets are described by Zhang et al. [1996]. Breese et al. [1998] provides an empirical analysis of different algorithms for collaborative filtering. Techniques for collaborative filtering of news articles are described by Konstan et al. [1997].

Chakrabarti [2000] provides a survey of hypertext mining techniques such as hypertext classification and clustering. Chakrabarti [1999] provides a survey of Web resource discovery. Techniques for integrating data cubes with data mining are described by Sarawagi [2000].

Poe [1995] and Mattison [1996] provide textbook coverage of data warehousing. Zhuge et al. [1995] describes view maintenance in a data-warehousing environment.

Witten et al. [1999], Grossman and Frieder [1998], and Baeza-Yates and Ribeiro-Neto [1999] provide textbook descriptions of information retrieval. Indexing of documents is covered in detail by Witten et al. [1999]. Jones and Willet [1997] is a collection of articles on information retrieval. Salton [1989] is an early textbook on information-retrieval systems. The TREC benchmark (trec.nist.gov) is a benchmark for measuring retrieval effectiveness.

Brin and Page [1998] describes the anatomy of the Google search engine, including the PageRank technique, while a hubs and authorities based ranking technique called HITS is described by Kleinberg [1999]. Bharat and Henzinger [1998] presents a refinement of the HITS ranking technique. A point worth noting is that the PageRank of a page is computed independent of any query, and as a result a highly ranked page which just happens to contain some irrelevant keywords would figure among the top answers for a query on the irrelevant keywords. In contrast, the HITS algorithm takes the query keywords into account when computing prestige, but has a higher cost for answering queries.

Tools

A variety of tools are available for each of the applications we have studied in this chapter. Most database vendors provide OLAP tools as part of their database system, or as add-on applications. These include OLAP tools from Microsoft Corp., Oracle Express, Informix Metacube. The Arbor Essbase OLAP tool is from an independent software vendor. The site www.databeacon.com provides an online demo of the databeacon OLAP tools for use on Web and text file data sources. Many companies also provide analysis tools specialized for specific applications, such as customer relationship management.

There is also a wide variety of general purpose data mining tools, including mining tools from the SAS Institute, IBM Intelligent Miner, and SGI Mineset. A good deal of expertise is required to apply general purpose mining tools for specific applications. As a result a large number of mining tools have been developed to address specialized applications. The Web site www.kdnuggets.com provides an extensive directory of mining software, solutions, publications, and so on.

Major database vendors also offer data warehousing products coupled with their database systems. These provide support functionality for data modeling, cleansing, loading, and querying. The Web site www.dwinfocenter.org provides information datawarehousing products.

Google (www.google.com) is a popular search engine. Yahoo (www.yahoo.com) and the Open Directory Project (dmoz.org) provide classification hierarchies for Web sites.

C H A P T E R 2 3

Advanced Data Types and New Applications

For most of the history of databases, the types of data stored in databases were relatively simple, and this was reflected in the rather limited support for data types in earlier versions of SQL. In the past few years, however, there has been increasing need for handling new data types in databases, such as temporal data, spatial data. and multimedia data.

Another major trend in the last decade has created its own issues: the growth of mobile computers, starting with laptop computers and pocket organizers, and in more recent years growing to include mobile phones with built-in computers, and a variety of *wearable* computers that are increasingly used in commercial applications.

In this chapter we study several new data types, and also study database issues dealing with mobile computers.

23.1 Motivation

Before we address each of the topics in detail, we summarize the motivation for, and some important issues in dealing with, each of these types of data.

- **Temporal data**. Most database systems model the current state of the world, for instance, current customers, current students, and courses currently being offered. In many applications, it is very important to store and retrieve information about past states. Historical information can be incorporated manually into a schema design. However, the task is greatly simplified by database support for temporal data, which we study in Section 23.2.

- **Spatial data**. Spatial data include **geographic data**, such as maps and associated information, and **computer-aided-design data**, such as integrated-circuit designs or building designs. Applications of spatial data initially stored data as files in a file system, as did early-generation business applications. But as the complexity and volume of the data, and the number of users, have grown, ad hoc approaches to storing and retrieving data in a file system have proved insufficient for the needs of many applications that use spatial data.

Spatial-data applications require facilities offered by a database system—in particular, the ability to store and query large amounts of data efficiently. Some applications may also require other database features, such as atomic updates to parts of the stored data, durability, and concurrency control. In Section 23.3, we study the extensions needed to traditional database systems to support spatial data.

- **Multimedia data**. In Section 23.4, we study the features required in database systems that store multimedia data such as image, video, and audio data. The main distinguishing feature of video and audio data is that the display of the data requires retrieval at a steady, predetermined rate; hence, such data are called **continuous-media data**.

- **Mobile databases**. In Section 23.5, we study the database requirements of the new generation of mobile computing systems, such as notebook computers and palmtop computing devices, which are connected to base stations via wireless digital communication networks. Such computers need to be able to operate while disconnected from the network, unlike the distributed database systems discussed in Chapter 19. They also have limited storage capacity, and thus require special techniques for memory management.

23.2 Time in Databases

A database models the state of some aspect of the real world outside itself. Typically, databases model only one state—the current state—of the real world, and do not store information about past states, except perhaps as audit trails. When the state of the real world changes, the database gets updated, and information about the old state gets lost. However, in many applications, it is important to store and retrieve information about past states. For example, a patient database must store information about the medical history of a patient. A factory monitoring system may store information about current and past readings of sensors in the factory, for analysis. Databases that store information about states of the real world across time are called **temporal databases**.

When considering the issue of time in database systems, we must distinguish between time as measured by the system and time as observed in the real world. The **valid time** for a fact is the set of time intervals during which the fact is true in the real world. The **transaction time** for a fact is the time interval during which the fact is current within the database system. This latter time is based on the transaction serialization order and is generated automatically by the system. Note that valid-time intervals, being a real-world concept, cannot be generated automatically and must be provided to the system.

A **temporal relation** is one where each tuple has an associated time when it is true; the time may be either valid time or transaction time. Of course, both valid time and transaction time can be stored, in which case the relation is said to be a

account-number	*branch-name*	*balance*	*from*	*to*
A-101	Downtown	500	1999/1/1 9:00	1999/1/24 11:30
A-101	Downtown	100	1999/1/24 11:30	*
A-215	Mianus	700	2000/6/2 15:30	2000/8/8 10:00
A-215	Mianus	900	2000/8/8 10:00	2000/9/5 8:00
A-215	Mianus	700	2000/9/5 8:00	*
A-217	Brighton	750	1999/7/5 11:00	2000/5/1 16:00

Figure 23.1 A temporal *account* relation.

bitemporal relation. Figure 23.1 shows an example of a temporal relation. To simplify the representation, each tuple has only one time interval associated with it; thus, a tuple is represented once for every disjoint time interval in which it is true. Intervals are shown here as a pair of attributes *from* and *to*; an actual implementation would have a structured type, perhaps called *Interval*, that contains both fields. Note that some of the tuples have a "*" in the *to* time column; these asterisks indicate that the tuple is true until the value in the *to* time column is changed; thus, the tuple is true at the current time. Although times are shown in textual form, they are stored internally in a more compact form, such as the number of seconds since some fixed time on a fixed date (such as 12:00 AM, January 1, 1900) that can be translated back to the normal textual form.

23.2.1 Time Specification in SQL

The SQL standard defines the types **date**, **time**, and **timestamp**. The type **date** contains four digits for the year (1−9999), two digits for the month (1−12), and two digits for the date (1−31). The type **time** contains two digits for the hour, two digits for the minute, and two digits for the second, plus optional fractional digits. The seconds field can go beyond 60, to allow for leap seconds that are added during some years to correct for small variations in the speed of rotation of Earth. The type **timestamp** contains the fields of **date** and **time**, with six fractional digits for the seconds field.

Since different places in the world have different local times, there is often a need for specifying the time zone along with the time. The **Universal Coordinated Time** (**UTC**), is a standard reference point for specifying time, with local times defined as offsets from UTC. (The standard abbreviation is UTC, rather than UCT, since it is an abbreviation of "Universal Coordinated Time" written in French as *universel temps coordonné*.) SQL also supports two types, **time with time zone**, and **timestamp with time zone**, which specify the time as a local time plus the offset of the local time from UTC. For instance, the time could be expressed in terms of U.S. Eastern Standard Time, with an offset of −6:00, since U.S. Eastern Standard time is 6 hours behind UTC.

SQL supports a type called **interval**, which allows us to refer to a period of time such as "1 day" or "2 days and 5 hours," without specifying a particular time when this period starts. This notion differs from the notion of interval we used previously, which refers to an interval of time with specific starting and ending times.[1]

23.2.2 Temporal Query Languages

A database relation without temporal information is sometimes called a **snapshot relation**, since it reflects the state in a snapshot of the real world. Thus, a snapshot of a temporal relation at a point in time t is the set of tuples in the relation that are true at time t, with the time-interval attributes projected out. The snapshot operation on a temporal relation gives the snapshot of the relation at a specified time (or the current time, if the time is not specified).

A **temporal selection** is a selection that involves the time attributes; a **temporal projection** is a projection where the tuples in the projection inherit their times from the tuples in the original relation. A **temporal join** is a join, with the time of a tuple in the result being the intersection of the times of the tuples from which it is derived. If the times do not intersect, the tuple is removed from the result.

The predicates *precedes*, *overlaps*, and *contains* can be applied on intervals; their meanings should be clear. The *intersect* operation can be applied on two intervals, to give a single (possibly empty) interval. However, the union of two intervals may or may not be a single interval.

Functional dependencies must be used with care in a temporal relation. Although the account number may functionally determine the balance at any given point in time, obviously the balance can change over time. A **temporal functional dependency** $X \xrightarrow{\tau} Y$ holds on a relation schema R if, for all legal instances r of R, all snapshots of r satisfy the functional dependency $X \rightarrow Y$.

Several proposals have been made for extending SQL to improve its support of temporal data. SQL:1999 Part 7 (SQL/Temporal), which is currently under development, is the proposed standard for temporal extensions to SQL.

23.3 Spatial and Geographic Data

Spatial data support in databases is important for efficiently storing, indexing, and querying of data based on spatial locations. For example, suppose that we want to store a set of polygons in a database, and to query the database to find all polygons that intersect a given polygon. We cannot use standard index structures, such as B-trees or hash indices, to answer such a query efficiently. Efficient processing of the above query would require special-purpose index structures, such as R-trees (which we study later) for the task.

Two types of spatial data are particularly important:

- **Computer-aided-design (CAD) data**, which includes spatial information about how objects—such as buildings, cars, or aircraft—are constructed. Other important examples of computer-aided-design databases are integrated-circuit and electronic-device layouts.

1. Many temporal database researchers feel this type should have been called **span** since it does not specify an exact start or end time, only the time span between the two.

- **Geographic data** such as road maps, land-usage maps, topographic elevation maps, political maps showing boundaries, land ownership maps, and so on. **Geographic information systems** are special-purpose databases tailored for storing geographic data.

Support for geographic data has been added to many database systems, such as the IBM DB2 Spatial Extender, the Informix Spatial Datablade, and Oracle Spatial.

23.3.1 Representation of Geometric Information

Figure 23.2 illustrates how various geometric constructs can be represented in a database, in a normalized fashion. We stress here that geometric information can be represented in several different ways, only some of which we describe.

A *line segment* can be represented by the coordinates of its endpoints. For example, in a map database, the two coordinates of a point would be its latitude and longi-

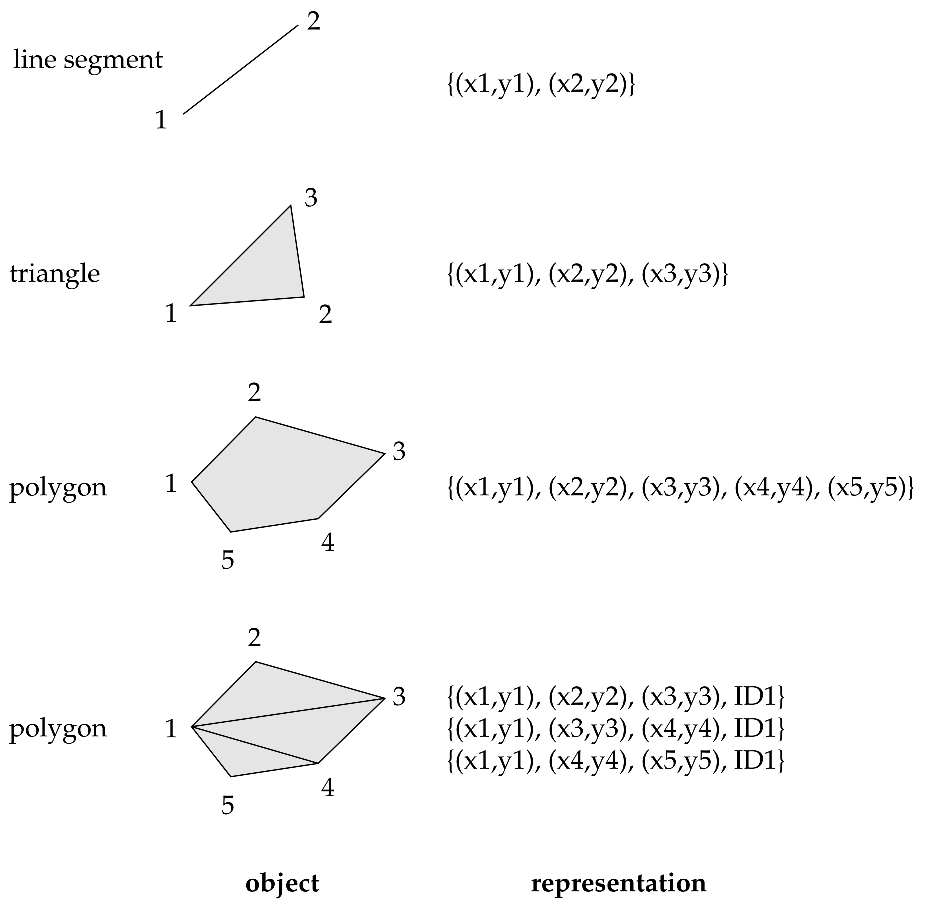

line segment
1
2
{(x1,y1), (x2,y2)}
triangle
1
2
3
{(x1,y1), (x2,y2), (x3,y3)}
polygon
1
2
3
4
5
{(x1,y1), (x2,y2), (x3,y3), (x4,y4), (x5,y5)}
polygon
1
2
3
4
5
{(x1,y1), (x2,y2), (x3,y3), ID1}
{(x1,y1), (x3,y3), (x4,y4), ID1}
{(x1,y1), (x4,y4), (x5,y5), ID1}
object
representation

Figure 23.2 Representation of geometric constructs.

tude. A *polyline* (also called a *linestring*) consists of a connected sequence of line segments, and can be represented by a list containing the coordinates of the endpoints of the segments, in sequence. We can approximately represent an arbitrary curve by polylines, by partitioning the curve into a sequence of segments. This representation is useful for two-dimensional features such as roads; here, the width of the road is small enough relative to the size of the full map that it can be considered two dimensional. Some systems also support *circular arcs* as primitives, allowing curves to be represented as sequences of arcs.

We can represent a *polygon* by listing its vertices in order, as in Figure 23.2.[2] The list of vertices specifies the boundary of a polygonal region. In an alternative representation, a polygon can be divided into a set of triangles, as shown in Figure 23.2. This process is called **triangulation**, and any polygon can be triangulated. The complex polygon can be given an identifier, and each of the triangles into which it is divided carries the identifier of the polygon. Circles and ellipses can be represented by corresponding types, or can be approximated by polygons.

List-based representations of polylines or polygons are often convenient for query processing. Such non-first-normal-form representations are used when supported by the underlying database. So that we can use fixed-size tuples (in first-normal form) for representing polylines, we can give the polyline or curve an identifier, and can represent each segment as a separate tuple that also carries with it the identifier of the polyline or curve. Similarly, the triangulated representation of polygons allows a first-normal-form relational representation of polygons.

The representation of points and line segments in three-dimensional space is similar to their representation in two-dimensional space, the only difference being that points have an extra z component. Similarly, the representation of planar figures—such as triangles, rectangles, and other polygons—does not change much when we move to three dimensions. Tetrahedrons and cuboids can be represented in the same way as triangles and rectangles. We can represent arbitrary polyhedra by dividing them into tetrahedrons, just as we triangulate polygons. We can also represent them by listing their faces, each of which is itself a polygon, along with an indication of which side of the face is inside the polyhedron.

23.3.2 Design Databases

Computer-aided-design (**CAD**) systems traditionally stored data in memory during editing or other processing, and wrote the data back to a file at the end of a session of editing. The drawbacks of such a scheme include the cost (programming complexity, as well as time cost) of transforming data from one form to another, and the need to read in an entire file even if only parts of it are required. For large designs, such as the design of a large-scale integrated circuit, or the design of an entire airplane, it may be impossible to hold the complete design in memory. Designers of object-oriented databases were motivated in large part by the database requirements of CAD

2. Some references use the term *closed polygon* to refer to what we call polygons, and refer to polylines as open polygons.

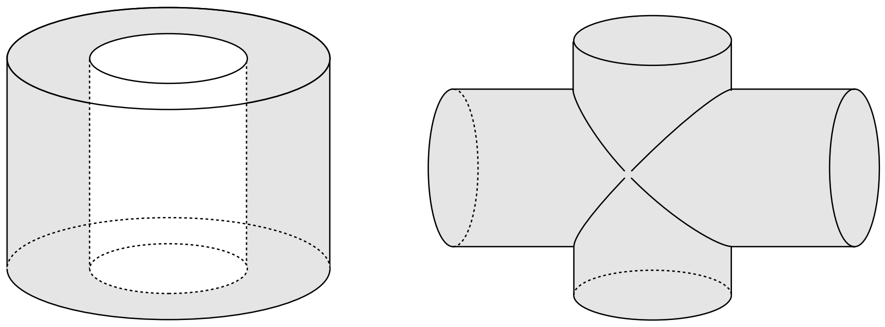
systems. Object-oriented databases represent components of the design as objects, and the connections between the objects indicate how the design is structured.

The objects stored in a design database are generally geometric objects. Simple two-dimensional geometric objects include points, lines, triangles, rectangles, and, in general, polygons. Complex two-dimensional objects can be formed from simple objects by means of union, intersection, and difference operations. Similarly, complex three-dimensional objects may be formed from simpler objects such as spheres, cylinders, and cuboids, by union, intersection, and difference operations, as in Figure 23.3. Three-dimensional surfaces may also be represented by **wireframe models**, which essentially model the surface as a set of simpler objects, such as line segments, triangles, and rectangles.

Design databases also store nonspatial information about objects, such as the material from which the objects are constructed. We can usually model such information by standard data-modeling techniques. We concern ourselves here with only the spatial aspects.

Various spatial operations must be performed on a design. For instance, the designer may want to retrieve that part of the design that corresponds to a particular region of interest. Spatial-index structures, discussed in Section 23.3.5, are useful for such tasks. Spatial-index structures are multidimensional, dealing with two- and three-dimensional data, rather than dealing with just the simple one-dimensional ordering provided by the B^+-trees.

Spatial-integrity constraints, such as "two pipes should not be in the same location," are important in design databases to prevent interference errors. Such errors often occur if the design is performed manually, and are detected only when a prototype is being constructed. As a result, these errors can be expensive to fix. Database support for spatial-integrity constraints helps people to avoid design errors, thereby keeping the design consistent. Implementing such integrity checks again depends on the availability of efficient multidimensional index structures.



(a) Difference of cylinders (b) Union of cylinders

Figure 23.3 Complex three-dimensional objects.

23.3.3 Geographic Data

Geographic data are spatial in nature, but differ from design data in certain ways. Maps and satellite images are typical examples of geographic data. Maps may provide not only location information—about boundaries, rivers, and roads, for example—but also much more detailed information associated with locations, such as elevation, soil type, land usage, and annual rainfall.

Geographic data can be categorized into two types:

- **Raster data**. Such data consist of bit maps or pixel maps, in two or more dimensions. A typical example of a two-dimensional raster image is a satellite image of cloud cover, where each pixel stores the cloud visibility in a particular area. Such data can be three-dimensional—for example, the temperature at different altitudes at different regions, again measured with the help of a satellite. Time could form another dimension—for example, the surface temperature measurements at different points in time. Design databases generally do not store raster data.

- **Vector data**. Vector data are constructed from basic geometric objects, such as points, line segments, triangles, and other polygons in two dimensions, and cylinders, spheres, cuboids, and other polyhedrons in three dimensions.

  Map data are often represented in vector format. Rivers and roads may be represented as unions of multiple line segments. States and countries may be represented as polygons. Topological information, such as height, may be represented by a surface divided into polygons covering regions of equal height, with a height value associated with each polygon.

23.3.3.1 Representation of Geographic Data

Geographical features, such as states and large lakes, are represented as complex polygons. Some features, such as rivers, may be represented either as complex curves or as complex polygons, depending on whether their width is relevant.

Geographic information related to regions, such as annual rainfall, can be represented as an array—that is, in raster form. For space efficiency, the array can be stored in a compressed form. In Section 23.3.5, we study an alternative representation of such arrays by a data structure called a *quadtree*.

As noted in Section 23.3.3, we can represent region information in vector form, using polygons, where each polygon is a region within which the array value is the same. The vector representation is more compact than the raster representation in some applications. It is also more accurate for some tasks, such as depicting roads, where dividing the region into pixels (which may be fairly large) leads to a loss of precision in location information. However, the vector representation is unsuitable for applications where the data are intrinsically raster based, such as satellite images.

23.3.3.2 Applications of Geographic Data

Geographic databases have a variety of uses, including online map services, vehicle-navigation systems; distribution-network information for public-service utilities such as telephone, electric-power, and water-supply systems; and land-usage information for ecologists and planners.

Web-based road map services form a very widely used application of map data. At the simplest level, these systems can be used to generate online road maps of a desired region. An important benefit of online maps is that it is easy to scale the maps to the desired size—that is, to zoom in and out to locate relevant features. Road map services also store information about roads and services, such as the layout of roads, speed limits on roads, road conditions, connections between roads, and one-way restrictions. With this additional information about roads, the maps can be used for getting directions to go from one place to another and for automatic trip planning. Users can query online information about services to locate, for example, hotels, gas stations, or restaurants with desired offerings and price ranges.

Vehicle-navigation systems are systems mounted in automobiles, which provide road maps and trip planning services. A useful addition to a mobile geographic information system such as a vehicle navigation system is a **Global Positioning System** (**GPS**) unit, which uses information broadcast from GPS satellites to find the current location with an accuracy of tens of meters. With such a system, a driver can never[3] get lost—the GPS unit finds the location in terms of latitude, longitude, and elevation and the navigation system can query the geographic database to find where and on which road the vehicle is currently located.

Geographic databases for public-utility information are becoming increasingly important as the network of buried cables and pipes grows. Without detailed maps, work carried out by one utility may damage the cables of another utility, resulting in large-scale disruption of service. Geographic databases, coupled with accurate location-finding systems, can help avoid such problems.

So far, we have explained why spatial databases are useful. In the rest of the section, we shall study technical details, such as representation and indexing of spatial information.

23.3.4 Spatial Queries

There are a number of types of queries that involve spatial locations.

- **Nearness queries** request objects that lie near a specified location. A query to find all restaurants that lie within a given distance of a given point is an example of a nearness query. The **nearest-neighbor query** requests the object that is nearest to a specified point. For example, we may want to find the nearest gasoline station. Note that this query does not have to specify a limit on the distance, and hence we can ask it even if we have no idea how far the nearest gasoline station lies.

- **Region queries** deal with spatial regions. Such a query can ask for objects that lie partially or fully inside a specified region. A query to find all retail shops within the geographic boundaries of a given town is an example.

3. Well, hardly ever!

- Queries may also request **intersections** and **unions** of regions. For example, given region information, such as annual rainfall and population density, a query may request all regions with a low annual rainfall as well as a high population density.

Queries that compute intersections of regions can be thought of as computing the **spatial join** of two spatial relations—for example, one representing rainfall and the other representing population density—with the location playing the role of join attribute. In general, given two relations, each containing spatial objects, the spatial join of the two relations generates either pairs of objects that intersect, or the intersection regions of such pairs.

Several join algorithms efficiently compute spatial joins on vector data. Although nested-loop join and indexed nested-loop join (with spatial indices) can be used, hash joins and sort–merge joins cannot be used on spatial data. Researchers have proposed join techniques based on coordinated traversal of spatial index structures on the two relations. See the bibliographical notes for more information.

In general, queries on spatial data may have a combination of spatial and nonspatial requirements. For instance, we may want to find the nearest restaurant that has vegetarian selections, and that charges less than $10 for a meal.

Since spatial data are inherently graphical, we usually query them by using a graphical query language. Results of such queries are also displayed graphically, rather than in tables. The user can invoke various operations on the interface, such as choosing an area to be viewed (for example, by pointing and clicking on suburbs west of Manhattan), zooming in and out, choosing what to display on the basis of selection conditions (for example, houses with more than three bedrooms), overlay of multiple maps (for example, houses with more than three bedrooms overlayed on a map showing areas with low crime rates), and so on. The graphical interface constitutes the front end. Extensions of SQL have been proposed to permit relational databases to store and retrieve spatial information efficiently, and also allowing queries to mix spatial and nonspatial conditions. Extensions include allowing abstract data types, such as lines, polygons, and bit maps, and allowing spatial conditions, such as *contains* or *overlaps*.

23.3.5 Indexing of Spatial Data

Indices are required for efficient access to spatial data. Traditional index structures, such as hash indices and B-trees, are not suitable, since they deal only with one-dimensional data, whereas spatial data are typically of two or more dimensions.

23.3.5.1 k-d Trees

To understand how to index spatial data consisting of two or more dimensions, we consider first the indexing of points in one-dimensional data. Tree structures, such as binary trees and B-trees, operate by successively dividing space into smaller parts. For instance, each internal node of a binary tree partitions a one-dimensional interval in two. Points that lie in the left partition go into the left subtree; points that lie in the right partition go into the right subtree. In a balanced binary tree, the partition is chosen so that approximately one-half of the points stored in the subtree fall in each partition. Similarly, each level of a B-tree splits a one-dimensional interval into multiple parts.

We can use that intuition to create tree structures for two-dimensional space, as well as in higher-dimensional spaces. A tree structure called a **k-d tree** was one of the early structures used for indexing in multiple dimensions. Each level of a k-d tree partitions the space into two. The partitioning is done along one dimension at the node at the top level of the tree, along another dimension in nodes at the next level, and so on, cycling through the dimensions. The partitioning proceeds in such a way that, at each node, approximately one-half of the points stored in the subtree fall on one side, and one-half fall on the other. Partitioning stops when a node has less than a given maximum number of points. Figure 23.4 shows a set of points in two-dimensional space, and a k-d tree representation of the set of points. Each line corresponds to a node in the tree, and the maximum number of points in a leaf node has been set at 1. Each line in the figure (other than the outside box) corresponds to a node in the k-d tree. The numbering of the lines in the figure indicates the level of the tree at which the corresponding node appears.

The **k-d-B tree** extends the k-d tree to allow multiple child nodes for each internal node, just as a B-tree extends a binary tree, to reduce the height of the tree. k-d-B trees are better suited for secondary storage than k-d trees.

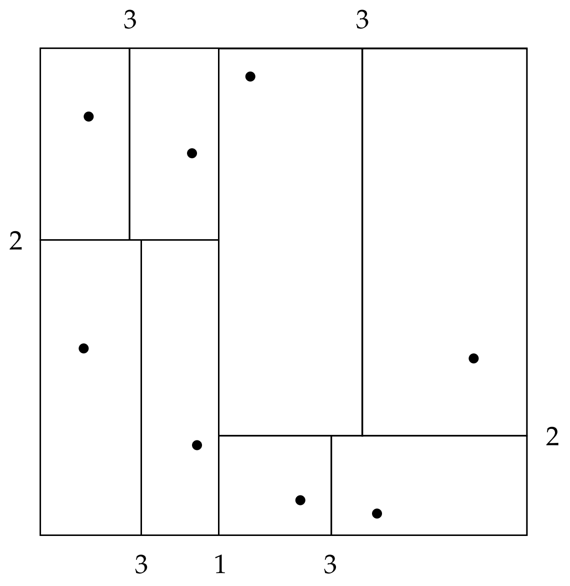


Figure 23.4 Division of space by a k-d tree.

23.3.5.2 Quadtrees

An alternative representation for two-dimensional data is a **quadtree**. An example of the division of space by a quadtree appears in Figure 23.5. The set of points is the same as that in Figure 23.4. Each node of a quadtree is associated with a rectangular region of space. The top node is associated with the entire target space. Each non-leaf node in a quadtree divides its region into four equal-sized quadrants, and correspondingly each such node has four child nodes corresponding to the four quadrants. Leaf nodes have between zero and some fixed maximum number of points. Correspondingly, if the region corresponding to a node has more than the maximum number of points, child nodes are created for that node. In the example in Figure 23.5, the maximum number of points in a leaf node is set to 1.

This type of quadtree is called a **PR quadtree**, to indicate it stores points, and that the division of space is divided based on regions, rather than on the actual set of points stored. We can use **region quadtrees** to store array (raster) information. A node in a region quadtree is a leaf node if all the array values in the region that it covers are the same. Otherwise, it is subdivided further into four children of equal area, and is therefore an internal node. Each node in the region quadtree corresponds to a subarray of values. The subarrays corresponding to leaves either contain just a single array element or have multiple array elements, all of which have the same value.

Indexing of line segments and polygons presents new problems. There are extensions of k-d trees and quadtrees for this task. However, a line segment or polygon may cross a partitioning line. If it does, it has to be split and represented in each of the subtrees in which its pieces occur. Multiple occurrences of a line segment or polygon can result in inefficiencies in storage, as well as inefficiencies in querying.

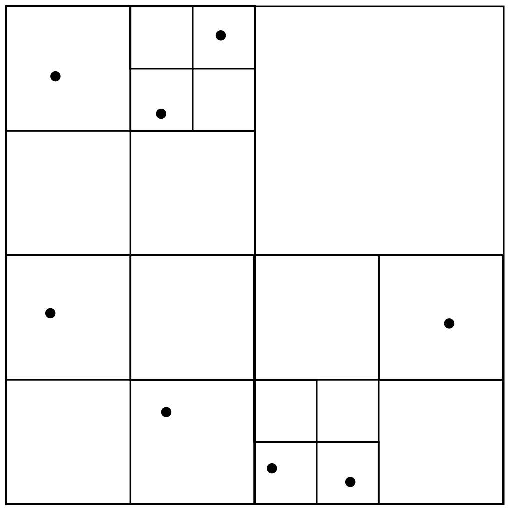


Figure 23.5 Division of space by a quadtree.

23.3.5.3 R-Trees

A storage structure called an **R-tree** is useful for indexing of rectangles and other polygons. An R-tree is a balanced tree structure with the indexed polygons stored in leaf nodes, much like a B^+-tree. However, instead of a range of values, a rectangular **bounding box** is associated with each tree node. The bounding box of a leaf node is the smallest rectangle parallel to the axes that contains all objects stored in the leaf node. The bounding box of internal nodes is, similarly, the smallest rectangle parallel to the axes that contains the bounding boxes of its child nodes. The bounding box of a polygon is defined, similarly, as the smallest rectangle parallel to the axes that contains the polygon.

Each internal node stores the bounding boxes of the child nodes along with the pointers to the child nodes. Each leaf node stores the indexed polygons, and may optionally store the bounding boxes of the polygons; the bounding boxes help speed up checks for overlaps of the rectangle with the indexed polygons—if a query rectangle does not overlap with the bounding box of a polygon, it cannot overlap with the polygon either. (If the indexed polygons are rectangles, there is of course no need to store bounding boxes since they are identical to the rectangles.)

Figure 23.6 shows an example of a set of rectangles (drawn with a solid line) and the bounding boxes (drawn with a dashed line) of the nodes of an R-tree for the set of rectangles. Note that the bounding boxes are shown with extra space inside them, to make them stand out pictorially. In reality, the boxes would be smaller and fit tightly on the objects that they contain; that is, each side of a bounding box B would touch at least one of the objects or bounding boxes that are contained in B.

The R-tree itself is at the right side of Figure 23.6. The figure refers to the coordinates of bounding box i as BB_i in the figure.

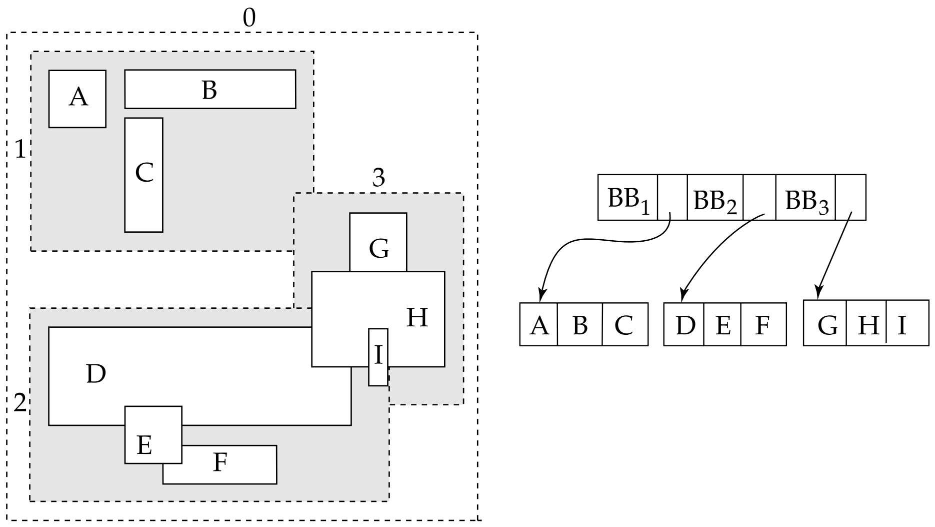


Figure 23.6 An R-tree.

We shall now see how to implement search, insert, and delete operations on an R-tree.

- **Search**: As the figure shows, the bounding boxes associated with sibling nodes may overlap; in B$^+$-trees, k-d trees, and quadtrees, in contrast, the ranges do not overlap. A search for polygons containing a point therefore has to follow *all* child nodes whose associated bounding boxes contain the point; as a result, multiple paths may have to be searched. Similarly, a query to find all polygons that intersect a given polygon has to go down every node where the associated rectangle intersects the polygon.

- **Insert**: When we insert a polygon into an R-tree, we select a leaf node to hold the polygon. Ideally we should pick a leaf node that has space to hold a new entry, and whose bounding box contains the bounding box of the polygon. However, such a node may not exist; even if it did, finding the node may be very expensive, since it is not possible to find it by a single traversal down from the root. At each internal node we may find multiple children whose bounding boxes contain the bounding box of the polygon, and each of these children needs to be explored. Therefore, as a heuristic, in a traversal from the root, if any of the child nodes has a bounding box containing the bounding box of the polygon, the R-tree algorithm chooses one of them arbitrarily. If none of the children satisfy this condition, the algorithm chooses a child node whose bounding box has the maximum overlap with the bounding box of the polygon for continuing the traversal.

  Once the leaf node has been reached, if the node is already full, the algorithm performs node splitting (and propagates splitting upward if required) in a manner very similar to B$^+$-tree insertion. Just as with B$^+$-tree insertion, the R-tree insertion algorithm ensures that the tree remains balanced. Additionally, it ensures that the bounding boxes of leaf nodes, as well as internal nodes, remain consistent; that is, bounding boxes of leaves contain all the bounding boxes of the polygons stored at the leaf, while the bounding boxes for internal nodes contain all the bounding boxes of the children nodes.

  The main difference of the insertion procedure from the B$^+$-tree insertion procedure lies in how the node is split. In a B$^+$-tree, it is possible to find a value such that half the entries are less than the midpoint and half are greater than the value. This property does not generalize beyond one dimension; that is, for more than one dimension, it is not always possible to split the entries into two sets so that their bounding boxes do not overlap. Instead, as a heuristic, the set of entries S can be split into two disjoint sets S_1 and S_2 so that the bounding boxes of S_1 and S_2 have the minimum total area; another heuristic would be to split the entries into two sets S_1 and S_2 in such a way that S_1 and S_2 have minimum overlap. The two nodes resulting from the split would contain the entries in S_1 and S_2 respectively. The cost of finding splits with minimum total area or overlap can itself be large, so cheaper heuristics, such as the *quadratic split* heuristic are used. (The heuristic gets is name from the fact that it takes time quadratic in the number of entries.)

The **quadratic split** heuristic works this way: First, it picks a pair of entries a and b from S such that putting them in the same node would result in a bounding box with the maximum wasted space; that is, the area of the minimum bounding box of a and b minus the sum of the areas of a and b is the largest. The heuristic places the entries a and b in sets S_1 and S_2 respectively.

It then iteratively adds the remaining entries, one entry per iteration, to one of the two sets S_1 or S_2. At each iteration, for each remaining entry e, let $i_{e,1}$ denote the increase in the size of the bounding box of S_1 if e is added to S_1 and let $i_{e,2}$ denote the corresponding increase for S_2. In each iteration, the heuristic chooses one of the entries with the maximum difference of $i_{e,1}$ and $i_{e,2}$ and adds it to S_1 if $i_{e,1}$ is less than $i_{e,2}$, and to S_2 otherwise. That is, an entry with "maximum preference" for one of S_1 or S_2 is chosen at each iteration. The iteration stops when all entries have been assigned, or when one of the sets S_1 or S_2 has enough entries that all remaining entries have to be added to the other set so the nodes constructed from S_1 and S_2 both have the required minimum occupancy. The heuristic then adds all unassigned entries to the set with fewer entries.

- **Deletion**: Deletion can be performed like a B$^+$-tree deletion, borrowing entries from sibling nodes, or merging sibling nodes if a node becomes underfull. An alternative approach redistributes all the entries of underfull nodes to sibling nodes, with the aim of improving the clustering of entries in the R-tree.

See the bibliographical references for more details on insertion and deletion operations on R-trees, as well as on variants of R-trees, called R*-trees or R^+-trees.

The storage efficiency of R-trees is better than that of k-d trees or quadtrees, since a polygon is stored only once, and we can ensure easily that each node is at least half full. However, querying may be slower, since multiple paths have to be searched. Spatial joins are simpler with quadtrees than with R-trees, since all quadtrees on a region are partitioned in the same manner. However, because of their better storage efficiency, and their similarity to B-trees, R-trees and their variants have proved popular in database systems that support spatial data.

23.4 Multimedia Databases

Multimedia data, such as images, audio, and video—an increasingly popular form of data—are today almost always stored outside the database, in file systems. This kind of storage is not a problem when the number of multimedia objects is relatively small, since features provided by databases are usually not important.

However, database features become important when the number of multimedia objects stored is large. Issues such as transactional updates, querying facilities, and indexing then become important. Multimedia objects often have descriptive attributes, such as those indicating when they were created, who created them, and to what category they belong. One approach to building a database for such multimedia objects is to use databases for storing the descriptive attributes and for keeping track of the files in which the multimedia objects are stored.

However, storing multimedia outside the database makes it harder to provide database functionality, such as indexing on the basis of actual multimedia data content. It can also lead to inconsistencies, such as a file that is noted in the database, but whose contents are missing, or vice versa. It is therefore desirable to store the data themselves in the database.

Several issues have to be addressed if multimedia data are to be stored in a database.

- The database must support large objects, since multimedia data such as videos can occupy up to a few gigabytes of storage. Many database systems do not support objects larger than a few gigabytes. Larger objects could be split into smaller pieces and stored in the database. Alternatively, the multimedia object may be stored in a file system, but the database may contain a pointer to the object; the pointer would typically be a file name. The SQL/MED standard (MED stands for Management of External Data), which is under development, allows external data, such as files, to be treated as if they are part of the database. With SQL/MED, the object would appear to be part of the database, but can be stored externally.

  We discuss multimedia data formats in Section 23.4.1.

- The retrieval of some types of data, such as audio and video, has the requirement that data delivery must proceed at a guaranteed steady rate. Such data are sometimes called **isochronous data**, or **continuous-media data**. For example, if audio data are not supplied in time, there will be gaps in the sound. If the data are supplied too fast, system buffers may overflow, resulting in loss of data. We discuss continuous-media data in Section 23.4.2.

- Similarity-based retrieval is needed in many multimedia database applications. For example, in a database that stores fingerprint images, a query fingerprint image is provided, and fingerprints in the database that are similar to the query fingerprint must be retrieved. Index structures such as B^+-trees and R-trees cannot be used for this purpose; special index structures need to be created. We discuss similarity-based retrieval in Section 23.4.3

23.4.1 Multimedia Data Formats

Because of the large number of bytes required to represent multimedia data, it is essential that multimedia data be stored and transmitted in compressed form. For image data, the most widely used format is *JPEG*, named after the standards body that created it, the *Joint Picture Experts Group*. We can store video data by encoding each frame of video in JPEG format, but such an encoding is wasteful, since successive frames of a video are often nearly the same. The *Moving Picture Experts Group* has developed the *MPEG* series of standards for encoding video and audio data; these encodings exploit commonalities among a sequence of frames to achieve a greater degree of compression. The *MPEG-1* standard stores a minute of 30-frame-per-second video and audio in approximately 12.5 megabytes (compared to approximately 75 megabytes for video in only JPEG). However, MPEG-1 encoding introduces some loss of video quality, to a level roughly comparable to that of VHS video tape.

The *MPEG-2* standard is designed for digital broadcast systems and digital video disks (DVD); it introduces only a negligible loss of video quality. MPEG-2 compresses 1 minute of video and audio to approximately 17 megabytes. Several competing standards are used for audio encoding, including *MP3*, which stands for MPEG-1 Layer 3, RealAudio, and other formats.

23.4.2 Continuous-Media Data

The most important types of continuous-media data are video and audio data (for example, a database of movies). Continuous-media systems are characterized by their real-time information-delivery requirements:

- Data must be delivered sufficiently fast that no gaps in the audio or video result.
- Data must be delivered at a rate that does not cause overflow of system buffers.
- Synchronization among distinct data streams must be maintained. This need arises, for example, when the video of a person speaking must show lips moving synchronously with the audio of the person speaking.

To supply data predictably at the right time to a large number of consumers of the data, the fetching of data from disk must be carefully coordinated. Usually, data are fetched in periodic cycles. In each cycle, say of n seconds, n seconds worth of data is fetched for each consumer and stored in memory buffers, while the data fetched in the previous cycle is being sent to the consumers from the memory buffers. The cycle period is a compromise: A short period uses less memory but requires more disk arm movement, which is a waste of resources, while a long period reduces disk arm movement but increases memory requirements and may delay initial delivery of data. When a new request arrives, **admission control** comes into play: That is, the system checks if the request can be satisfied with available resources (in each period); if so, it is admitted; otherwise it is rejected.

Extensive research on delivery of continuous media data has dealt with such issues as handling arrays of disks and dealing with disk failure. See the bibliographical references for details.

Several vendors offer video-on-demand servers. Current systems are based on file systems, because existing database systems do not provide the real-time response that these applications need. The basic architecture of a video-on-demand system comprises:

- **Video server**. Multimedia data are stored on several disks (usually in a RAID configuration). Systems containing a large volume of data may use tertiary storage for less frequently accessed data.
- **Terminals.** People view multimedia data through various devices, collectively referred to as *terminals*. Examples are personal computers and televisions attached to a small, inexpensive computer called a **set-top box**.

- **Network.** Transmission of multimedia data from a server to multiple terminals requires a high-capacity network.

Video-on-demand service eventually will become ubiquitous, just as cable and broadcast television are now. For the present, the main applications of video-server technology are in offices (for training, viewing recorded talks and presentations, and the like), in hotels, and in video-production facilities.

23.4.3 Similarity-Based Retrieval

In many multimedia applications, data are described only approximately in the database. An example is the fingerprint data in Section 23.4. Other examples are:

- **Pictorial data.** Two pictures or images that are slightly different as represented in the database may be considered the same by a user. For instance, a database may store trademark designs. When a new trademark is to be registered, the system may need first to identify all similar trademarks that were registered previously.
- **Audio data.** Speech-based user interfaces are being developed that allow the user to give a command or identify a data item by speaking. The input from the user must then be tested for similarity to those commands or data items stored in the system.
- **Handwritten data.** Handwritten input can be used to identify a handwritten data item or command stored in the database. Here again, similarity testing is required.

The notion of similarity is often subjective and user specific. However, similarity testing is often more successful than speech or handwriting recognition, because the input can be compared to data already in the system and, thus, the set of choices available to the system is limited.

Several algorithms exist for finding the best matches to a given input by similarity testing. Some systems, including a dial-by-name, voice-activated telephone system, have been deployed commercially. See the bibliographical notes for references.

23.5 Mobility and Personal Databases

Large-scale, commercial databases have traditionally been stored in central computing facilities. In distributed database applications, there has usually been strong central database and network administration. Two technology trends have combined to create applications in which this assumption of central control and administration is not entirely correct:

1. The increasingly widespread use of personal computers, and, more important, of laptop or notebook computers.

2. The development of a relatively low-cost wireless digital communication infrastructure, based on wireless local-area networks, cellular digital packet networks, and other technologies.

Mobile computing has proved useful in many applications. Many business travelers use laptop computers so that they can work and access data en route. Delivery services use mobile computers to assist in package tracking. Emergency-response services use mobile computers at the scene of disasters, medical emergencies, and the like to access information and to enter data pertaining to the situation. New applications of mobile computers continue to emerge.

Wireless computing creates a situation where machines no longer have fixed locations and network addresses. **Location-dependent queries** are an interesting class of queries that are motivated by mobile computers; in such queries, the location of the user (computer) is a parameter of the query. The value of the location parameter is provided either by the user or, increasingly, by a global positioning system (GPS). An example is a traveler's information system that provides data on hotels, roadside services, and the like to motorists. Processing of queries about services that are ahead on the current route must be based on knowledge of the user's location, direction of motion, and speed. Increasingly, navigational aids are being offered as a built-in feature in automobiles.

Energy (battery power) is a scarce resource for most mobile computers. This limitation influences many aspects of system design. Among the more interesting consequences of the need for energy efficiency is the use of scheduled data broadcasts to reduce the need for mobile systems to transmit queries.

Increasing amounts of data may reside on machines administered by users, rather than by database administrators. Furthermore, these machines may, at times, be disconnected from the network. In many cases, there is a conflict between the user's need to continue to work while disconnected and the need for global data consistency.

In Sections 23.5.1 through 23.5.4, we discuss techniques in use and under development to deal with the problems of mobility and personal computing.

23.5.1 A Model of Mobile Computing

The mobile-computing environment consists of mobile computers, referred to as **mobile hosts**, and a wired network of computers. Mobile hosts communicate with the wired network via computers referred to as **mobile support stations**. Each mobile support station manages those mobile hosts within its **cell**—that is, the geographical area that it covers. Mobile hosts may move between cells, thus necessitating a **handoff** of control from one mobile support station to another. Since mobile hosts may, at times, be powered down, a host may leave one cell and rematerialize later at some distant cell. Therefore, moves between cells are not necessarily between adjacent cells. Within a small area, such as a building, mobile hosts may be connected by a wireless local-area network (LAN) that provides lower-cost connectivity than would a wide-area cellular network, and that reduces the overhead of handoffs.

It is possible for mobile hosts to communicate directly without the intervention of a mobile support station. However, such communication can occur between only nearby hosts. Such direct forms of communication are becoming more prevalent with the advent of the **Bluetooth** standard. Bluetooth uses short-range digital radio to allow wireless connectivity within a 10-meter range at high speed (up to 721 kilobits per second). Initially conceived as a replacement for cables, Bluetooth's greatest promise is in easy ad hoc connection of mobile computers, PDAs, mobile phones, and so-called intelligent appliances.

The network infrastructure for mobile computing consists in large part of two technologies: wireless local-area networks (such as Avaya's Orinoco wireless LAN), and packet-based cellular telephony networks. Early cellular systems used analog technology and were designed for voice communication. Second-generation digital systems retained the focus on voice appliations. Third-generation (3G) and so-called 2.5G systems use packet-based networking and are more suited to data applications. In these networks, voice is just one of many applications (albeit an economically important one).

Bluetooth, wireless LANs, and 2.5G and 3G cellular networks make it possible for a wide variety of devices to communicate at low cost. While such communication itself does not fit the domain of a usual database application, the accounting, monitoring, and management data pertaining to this communication will generate huge databases. The immediacy of wireless communication generates a need for real-time access to many of these databases. This need for timeliness adds another dimension to the constraints on the system—a matter we shall discuss further in Section 24.3.

The size and power limitations of many mobile computers have led to alternative memory hierarchies. Instead of, or in addition to, disk storage, flash memory, which we discussed in Section 11.1, may be included. If the mobile host includes a hard disk, the disk may be allowed to spin down when it is not in use, to save energy. The same considerations of size and energy limit the type and size of the display used in a mobile device. Designers of mobile devices often create special-purpose user interfaces to work within these constraints. However, the need to present Web-based data has neccessitated the creation of presentation standards. **Wireless application protocol** (WAP) is a standard for wireless internet access. WAP-based browsers access special Web pages that use **wireless markup lanaguge** (WML), an XML-based language designed for the constraints of mobile and wireless Web browsing.

23.5.2 Routing and Query Processing

The route between a pair of hosts may change over time if one of the two hosts is mobile. This simple fact has a dramatic effect at the network level, since location-based network addresses are no longer constants within the system.

Mobility also directly affects database query processing. As we saw in Chapter 19, we must consider the communication costs when we choose a distributed query-processing strategy. Mobility results in dynamically changing communication costs, thus complicating the optimization process. Furthermore, there are competing notions of cost to consider:

- **User time** is a highly valuable commodity in many business applications
- **Connection time** is the unit by which monetary charges are assigned in some cellular systems
- **Number of bytes, or packets, transferred** is the unit by which charges are computed in some digital cellular systems
- **Time-of-day-based charges** vary, depending on whether communication occurs during peak or off-peak periods
- **Energy** is limited. Often, battery power is a scarce resource whose use must be optimized. A basic principle of radio communication is that it requires less energy to receive than to transmit radio signals. Thus, transmission and reception of data impose different power demands on the mobile host.

23.5.3 Broadcast Data

It is often desirable for frequently requested data to be broadcast in a continuous cycle by mobile support stations, rather than transmitted to mobile hosts on demand. A typical application of such **broadcast data** is stock-market price information. There are two reasons for using broadcast data. First, the mobile host avoids the energy cost for transmitting data requests. Second, the broadcast data can be received by a large number of mobile hosts at once, at no extra cost. Thus, the available transmission bandwidth is utilized more effectively.

A mobile host can then receive data as they are transmitted, rather than consuming energy by transmitting a request. The mobile host may have local nonvolatile storage available to cache the broadcast data for possible later use. Given a query, the mobile host may optimize energy costs by determining whether it can process that query with only cached data. If the cached data are insufficient, there are two options: Wait for the data to be broadcast, or transmit a request for data. To make this decision, the mobile host must know when the relevant data will be broadcast.

Broadcast data may be transmitted according to a fixed schedule or a changeable schedule. In the former case, the mobile host uses the known fixed schedule to determine when the relevant data will be transmitted. In the latter case, the broadcast schedule must itself be broadcast at a well-known radio frequency and at well-known time intervals.

In effect, the broadcast medium can be modeled as a disk with a high latency. Requests for data can be thought of as being serviced when the requested data are broadcast. The transmission schedules behave like indices on the disk. The bibliographical notes list recent research papers in the area of broadcast data management.

23.5.4 Disconnectivity and Consistency

Since wireless communication may be paid for on the basis of connection time, there is an incentive for certain mobile hosts to be disconnected for substantial periods. Mobile computers without wireless connectivity are disconnected most of the time when they are being used, except periodically when they are connected to their host computers, either physically or through a computer network.

During these periods of disconnection, the mobile host may remain in operation. The user of the mobile host may issue queries and updates on data that reside or are cached locally. This situation creates several problems, in particular:

- **Recoverability**: Updates entered on a disconnected machine may be lost if the mobile host experiences a catastrophic failure. Since the mobile host represents a single point of failure, stable storage cannot be simulated well.
- **Consistency**: Locally cached data may become out of date, but the mobile host cannot discover this situation until it is reconnected. Likewise, updates occurring in the mobile host cannot be propagated until reconnection occurs.

We explored the consistency problem in Chapter 19, where we discussed network partitioning, and we elaborate on it here. In wired distributed systems, partitioning is considered to be a failure mode; in mobile computing, partitioning via disconnection is part of the normal mode of operation. It is therefore necessary to allow data access to proceed despite partitioning, even at the risk of some loss of consistency.

For data updated by only the mobile host, it is a simple matter to propagate the updates when the mobile host reconnects. However, if the mobile host caches read-only copies of data that may be updated by other computers, the cached data may become inconsistent. When the mobile host is connected, it can be sent **invalidation reports** that inform it of out-of-date cache entries. However, when the mobile host is disconnected, it may miss an invalidation report. A simple solution to this problem is to invalidate the entire cache on reconnection, but such an extreme solution is highly costly. Several caching schemes are cited in the bibliographical notes.

If updates can occur at both the mobile host and elsewhere, detecting conflicting updates is more difficult. **Version-numbering**-based schemes allow updates of shared files from disconnected hosts. These schemes do not guarantee that the updates will be consistent. Rather, they guarantee that, if two hosts independently update the same version of a document, the clash will be detected eventually, when the hosts exchange information either directly or through a common host.

The **version-vector scheme** detects inconsistencies when copies of a document are independently updated. This scheme allows copies of a *document* to be stored at multiple hosts. Although we use the term *document*, the scheme can be applied to any other data items, such as tuples of a relation.

The basic idea is for each host i to store, with its copy of each document d, a **version vector**—that is, a set of version numbers $\{V_{d,i}[j]\}$, with one entry for each other host j on which the document could potentially be updated. When a host i updates a document d, it increments the version number $V_{d,i}[i]$ by one.

Whenever two hosts i and j connect with each other, they exchange updated documents, so that both obtain new versions of the documents. However, before exchanging documents, the hosts have to discover whether the copies are consistent:

1. If the version vectors are the same on both hosts—that is, for each k, $V_{d,i}[k] = V_{d,j}[k]$—then the copies of document d are identical.

2. If, for each k, $V_{d,i}[k] \leq V_{d,j}[k]$ and the version vectors are not identical, then the copy of document d at host i is older than the one at host j. That is, the copy of document d at host j was obtained by one or more modifications of the copy of the document at host i. Host i replaces its copy of d, as well as its copy of the version vector for d, with the copies from host j.
3. If there is a pair of hosts k and m such that $V_{d,i}[k] < V_{d,j}[k]$ and $V_{d,i}[m] > V_{d,j}[m]$, then the copies are *inconsistent*; that is, the copy of d at i contains updates performed by host k that have not been propagated to host j, and, similarly, the copy of d at j contains updates performed by host m that have not been propagated to host i. Then, the copies of d are inconsistent, since two or more updates have been performed on d independently. Manual intervention may be required to merge the updates.

The version-vector scheme was initially designed to deal with failures in distributed file systems. The scheme gained importance because mobile computers often store copies of files that are also present on server systems, in effect constituting a distributed file system that is often disconnected. Another application of the scheme is in groupware systems, where hosts are connected periodically, rather than continuously, and must exchange updated documents. The version-vector scheme also has applications in replicated databases.

The version-vector scheme, however, fails to address the most difficult and most important issue arising from updates to shared data—the reconciliation of inconsistent copies of data. Many applications can perform reconciliation automatically by executing in each computer those operations that had performed updates on remote computers during the period of disconnection. This solution works if update operations commute—that is, they generate the same result, regardless of the order in which they are executed. Alternative techniques may be available in certain applications; in the worst case, however, it must be left to the users to resolve the inconsistencies. Dealing with such inconsistency automatically, and assisting users in resolving inconsistencies that cannot be handled automatically, remains an area of research.

Another weakness is that the version-vector scheme requires substantial communication between a reconnecting mobile host and that host's mobile support station. Consistency checks can be delayed until the data are needed, although this delay may increase the overall inconsistency of the database.

The potential for disconnection and the cost of wireless communication limit the practicality of transaction-processing techniques discussed in Chapter 19 for distributed systems. Often, it is preferable to let users prepare transactions on mobile hosts, but to require that, instead of executing the transactions locally, they submit transactions to a server for execution. Transactions that span more than one computer and that include a mobile host face long-term blocking during transaction commit, unless disconnectivity is rare or predictable.

23.6 Summary

- Time plays an important role in database systems. Databases are models of the real world. Whereas most databases model the state of the real world at a point in time (at the current time), temporal databases model the states of the real world across time.

- Facts in temporal relations have associated times when they are valid, which can be represented as a union of intervals. Temporal query languages simplify modeling of time, as well as time-related queries.
- Spatial databases are finding increasing use today to store computer-aided-design data as well as geographic data.
- Design data are stored primarily as vector data; geographic data consist of a combination of vector and raster data. Spatial-integrity constraints are important for design data.
- Vector data can be encoded as first-normal-form data, or can be stored using non-first-normal-form structures, such as lists. Special-purpose index structures are particularly important for accessing spatial data, and for processing spatial queries.
- R-trees are a multidimensional extension of B-trees; with variants such as R+-trees and R*-trees, they have proved popular in spatial databases. Index structures that partition space in a regular fashion, such as quadtrees, help in processing spatial join queries.
- Multimedia databases are growing in importance. Issues such as similarity-based retrieval and delivery of data at guaranteed rates are topics of current research.
- Mobile computing systems have become common, leading to interest in database systems that can run on such systems. Query processing in such systems may involve lookups on server databases. The query cost model must include the cost of communication, including monetary cost and battery-power cost, which is relatively high for mobile systems.
- Broadcast is much cheaper per recipient than is point-to-point communication, and broadcast of data such as stock-market data helps mobile systems to pick up data inexpensively.
- Disconnected operation, use of broadcast data, and caching of data are three important issues being addressed in mobile computing.

Review Terms

- Temporal data
- Valid time
- Transaction time
- Temporal relation
- Bitemporal relation
- Universal coordinated time (UTC)
- Snapshot relation
- Temporal query languages
- Temporal selection
- Temporal projection

- Temporal join
- Spatial and geographic data
- Computer-aided-design (CAD) data
- Geographic data
- Geographic information systems
- Triangulation
- Design databases
- Geographic data
- Raster data
- Vector data
- Global positioning system (GPS)
- Spatial queries
- Nearness queries
- Nearest-neighbor queries
- Region queries
- Spatial join
- Indexing of spatial data
- k-d trees
- k-d-B trees
- Quadtrees
  - ☐ PR quadtree
  - ☐ Region quadtree
- R-trees
  - ☐ Bounding box
  - ☐ Quadratic split
- Multimedia databases
- Isochronous data
- Continuous-media data
- Similarity-based retrieval
- Multimedia data formats
- Video servers
- Mobile computing
  - ☐ Mobile hosts
  - ☐ Mobile support stations
  - ☐ Cell
  - ☐ Handoff
- Location-dependent queries
- Broadcast data
- Consistency
  - ☐ Invalidation reports
  - ☐ Version-vector scheme

Exercises

23.1 What are the two types of time, and how are they different? Why does it make sense to have both types of time associated with a tuple?

23.2 Will functional dependencies be preserved if a relation is converted to a temporal relation by adding a time attribute? How is the problem handled in a temporal database?

23.3 Suppose you have a relation containing the x, y coordinates and names of restaurants. Suppose also that the only queries that will be asked are of the following form: The query specifies a point, and asks if there is a restaurant exactly at that point. Which type of index would be preferable, R-tree or B-tree? Why?

23.4 Consider two-dimensional vector data where the data items do not overlap. Is it possible to convert such vector data to raster data? If so, what are the drawbacks of storing raster data obtained by such conversion, instead of the original vector data?

23.5 Suppose you have a spatial database that supports region queries (with circular regions) but not nearest-neighbor queries. Describe an algorithm to find the nearest neighbor by making use of multiple region queries.

23.6 Suppose you want to store line segments in an R-tree. If a line segment is not parallel to the axes, the bounding box for it can be large, containing a large empty area.

- Describe the effect on performance of having large bounding boxes on queries that ask for line segments intersecting a given region.
- Briefly describe a technique to improve performance for such queries and give an example of its benefit. Hint: you can divide segments into smaller pieces.

23.7 Give a recursive procedure to efficiently compute the spatial join of two relations with R-tree indices. (Hint: Use bounding boxes to check if leaf entries under a pair of internal nodes may intersect.)

23.8 Study the support for spatial data offered by the database system that you use, and implement the following:

a. A schema to represent the geographic location of restaurants along with features such as the cuisine served at the restaurant and the level of expensiveness.
b. A query to find moderately priced restaurants that serve Indian food and are within 5 miles of your house (assume any location for your house).
c. A query to find for each restaurant the distance from the nearest restaurant serving the same cuisine and with the same level of expensiveness.

23.9 What problems can occur in a continuous-media system if data is delivered either too slowly or too fast?

23.10 Describe how the ideas behind the RAID organization (Section 11.3) can be used in a broadcast-data environment, where there may occasionally be noise that prevents reception of part of the data being transmitted.

23.11 List three main features of mobile computing over wireless networks that are distinct from traditional distributed systems.

23.12 List three factors that need to be considered in query optimization for mobile computing that are not considered in traditional query optimizers.

23.13 Define a model of repeatedly broadcast data in which the broadcast medium is modeled as a virtual disk. Describe how access time and data-transfer rate for this virtual disk differ from the corresponding values for a typical hard disk.

23.14 Consider a database of documents in which all documents are kept in a central database. Copies of some documents are kept on mobile computers. Suppose that mobile computer A updates a copy of document 1 while it is disconnected, and, at the same time, mobile computer B updates a copy of document 2 while it is disconnected. Show how the version-vector scheme can ensure proper updating of the central database and mobile computers when a mobile computer reconnects.

23.15 Give an example to show that the version-vector scheme does not ensure serializability. (Hint: Use the example from Exercise 23.14, with the assumption that documents 1 and 2 are available on both mobile computers A and B, and take into account the possibility that a document may be read without being updated.)

Bibliographical Notes

The incorporation of time into the relational data model is discussed in Snodgrass and Ahn [1985], Clifford and Tansel [1985], Gadia [1986], Gadia [1988], Snodgrass [1987], Tansel et al. [1993], Snodgrass et al. [1994], and Tuzhilin and Clifford [1990]. Stam and Snodgrass [1988] and Soo [1991] provide surveys on temporal data management. Jensen et al. [1994] presents a glossary of temporal-database concepts, aimed at unifying the terminology. a proposal that had significant impact on the SQL standard. Tansel et al. [1993] is a collection of articles on different aspects of temporal databases. Chomicki [1995] presents techniques for managing temporal integrity constraints. A concept of completeness for temporal query languages analogous to relational completeness (equivalence to the relational algebra) is given in Clifford et al. [1994].

Samet [1995b] provides an overview of the large amount of work on spatial index structures. Samet [1990] provides a textbook coverage of spatial data structures. An early description of the quad tree is provided by Finkel and Bentley [1974]. Samet [1990] and Samet [1995b] describe numerous variants of quad trees. Bentley [1975] describes the k-d tree, and Robinson [1981] describes the k-d-B tree. The R-tree was originally presented in Guttman [1984]. Extensions of the R-tree are presented by Sellis et al. [1987], which describes the R^+ tree; Beckmann et al. [1990], which describes the R^* tree; and Kamel and Faloutsos [1992], which describes a parallel version of the R-tree.

Brinkhoff et al. [1993] discusses an implementation of spatial joins using R-trees. Lo and Ravishankar [1996] and Patel and DeWitt [1996] present partitioning-based methods for computation of spatial joins. Samet and Aref [1995] provides an overview of spatial data models, spatial operations, and the integration of spatial and nonspatial data. Indexing of handwritten documents is discussed in Aref et al. [1995b], Aref et al. [1995a], and Lopresti and Tomkins [1993]. Joins of approximate data are discussed in Barbará et al. [1992]. Evangelidis et al. [1995] presents a technique for concurrent access to indices on spatial data.

Samet [1995a] describes research issues in multimedia databases. Indexing of multimedia data is discussed in Faloutsos and Lin [1995]. Video servers are discussed in Anderson et al. [1992], Rangan et al. [1992], Ozden et al. [1994], Freedman and DeWitt [1995], and Ozden et al. [1996b]. Fault tolerance is discussed in Berson et al. [1995] and Ozden et al. [1996a]. Reason et al. [1996] suggests alternative compression schemes for video transmission over wireless networks. Disk storage management techniques for video data are described in Chen et al. [1995], Chervenak et al. [1995], Ozden et al. [1995a], and Ozden et al. [1995b].

Information management in systems that include mobile computers is studied in Alonso and Korth [1993] and Imielinski and Badrinath [1994]. Imielinski and Korth [1996] presents an introduction to mobile computing and a collection of research papers on the subject. Indexing of data broadcast over wireless media is considered in Imielinski et al. [1995]. Caching of data in mobile environments is discussed in Barbará and Imielinski [1994] and Acharya et al. [1995]. Disk management in mobile computers is addressed in Douglis et al. [1994].

The version-vector scheme for detecting inconsistency in distributed file systems is described by Popek et al. [1981] and Parker et al. [1983].

CHAPTER 24

Advanced Transaction Processing

In Chapters 15, 16, and 17, we introduced the concept of a transaction, which is a program unit that accesses—and possibly updates—various data items, and whose execution ensures the preservation of the ACID properties. We discussed in those chapters a variety of schemes for ensuring the ACID properties in an environment where failure can occur, and where the transactions may run concurrently.

In this chapter, we go beyond the basic schemes discussed previously, and cover advanced transaction-processing concepts, including transaction-processing monitors, transactional workflows, main-memory databases, real-time databases, long-duration transactions, nested transactions, and multidatabase transactions.

24.1 Transaction-Processing Monitors

Transaction-processing monitors (**TP monitors**) are systems that were developed in the 1970s and 1980s, initially in response to a need to support a large number of remote terminals (such as airline-reservation terminals) from a single computer. The term *TP monitor* initially stood for *teleprocessing monitor*.

TP monitors have since evolved to provide the core support for distributed transaction processing, and the term TP monitor has acquired its current meaning. The CICS TP monitor from IBM was one of the earliest TP monitors, and has been very widely used. Current-generation TP monitors include Tuxedo and Top End (both now from BEA Systems), Encina (from Transarc, which is now a part of IBM), and Transaction Server (from Microsoft).

24.1.1 TP-Monitor Architectures

Large-scale transaction processing systems are built around a client–server architecture. One way of building such systems is to have a server process for each client; the server performs authentication, and then executes actions requested by the client.

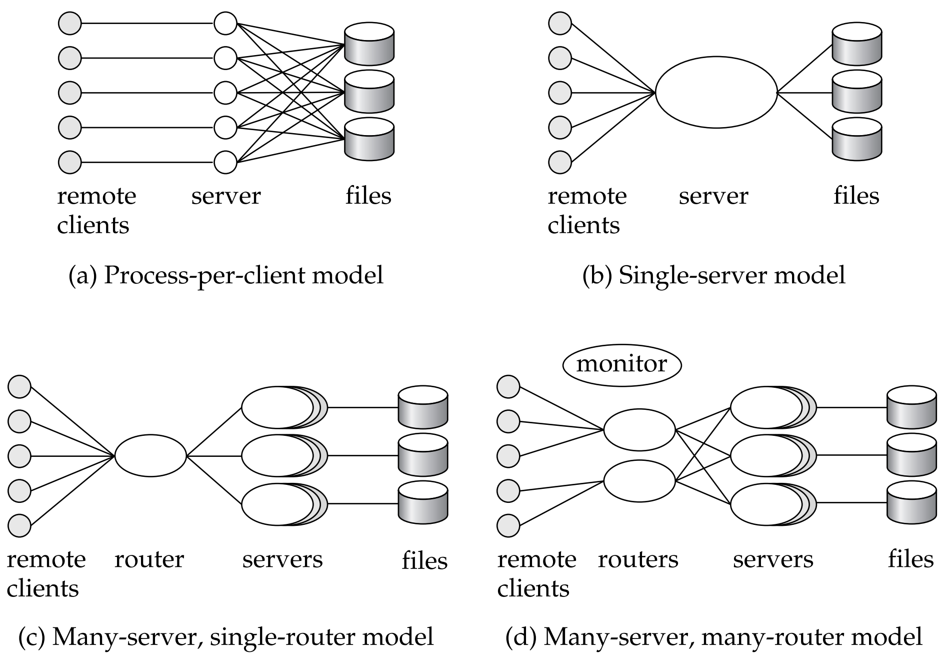


Figure 24.1 TP-monitor architectures.

This **process-per-client model** is illustrated in Figure 24.1a. This model presents several problems with respect to memory utilization and processing speed:

- Per-process memory requirements are high. Even if memory for program code is shared by all processes, each process consumes memory for local data and open file descriptors, as well as for operating-system overhead, such as page tables to support virtual memory.
- The operating system divides up available CPU time among processes by switching among them; this technique is called **multitasking**. Each **context switch** between one process and the next has considerable CPU overhead; even on today's fast systems, a context switch can take hundreds of microseconds.

The above problems can be avoided by having a single-server process to which all remote clients connect; this model is called the **single-server model**, illustrated in Figure 24.1b. Remote clients send requests to the server process, which then executes those requests. This model is also used in client–server environments, where clients send requests to a single-server process. The server process handles tasks, such as user authentication, that would normally be handled by the operating system. To avoid blocking other clients when processing a long request for one client, the server process is **multithreaded**: The server process has a thread of control for each client, and, in effect, implements its own low-overhead multitasking. It executes code on behalf of one client for a while, then saves the internal context and switches to the code for another client. Unlike the overhead of full multitasking, the cost of switching between threads is low (typically only a few microseconds).

Systems based on the single-server model, such as the original version of the IBM CICS TP monitor and file servers such as Novel's NetWare, successfully provided high transaction rates with limited resources. However, they had problems, especially when multiple applications accessed the same database:

- Since all the applications run as a single process, there is no protection among them. A bug in one application can affect all the other applications as well. It would be best to run each application as a separate process.
- Such systems are not suited for parallel or distributed databases, since a server process cannot execute on multiple computers at once. (However, concurrent threads within a process can be supported in a shared-memory multiprocessor system.) This is a serious drawback in large organizations, where parallel processing is critical for handling large workloads, and distributed data are becoming increasingly common.

One way to solve these problems is to run multiple application-server processes that access a common database, and to let the clients communicate with the application through a single communication process that routes requests. This model is called the **many-server, single-router model**, illustrated in Figure 24.1c. This model supports independent server processes for multiple applications; further, each application can have a pool of server processes, any one of which can handle a client session. The request can, for example, be routed to the most lightly loaded server in a pool. As before, each server process can itself be multithreaded, so that it can handle multiple clients concurrently. As a further generalization, the application servers can run on different sites of a parallel or distributed database, and the communication process can handle the coordination among the processes.

The above architecture is also widely used in Web servers. A Web server has a main process that receives HTTP requests, and then assigns the task of handling each request to a separate process (chosen from among a pool of processes). Each of the processes is itself multithreaded, so that it can handle multiple requests.

A more general architecture has multiple processes, rather than just one, to communicate with clients. The client communication processes interact with one or more router processes, which route their requests to the appropriate server. Later-generation TP monitors therefore have a different architecture, called the **many-server, many-router model**, illustrated in Figure 24.1d. A controller process starts up the other processes, and supervises their functioning. Tandem Pathway is an example of the later-generation TP monitors that use this architecture. Very high performance Web server systems also adopt such an architecture.

The detailed structure of a TP monitor appears in Figure 24.2. A TP monitor does more than simply pass messages to application servers. When messages arrive, they may have to be queued; thus, there is a **queue manager** for incoming messages. The queue may be a **durable queue**, whose entries survive system failures. Using a

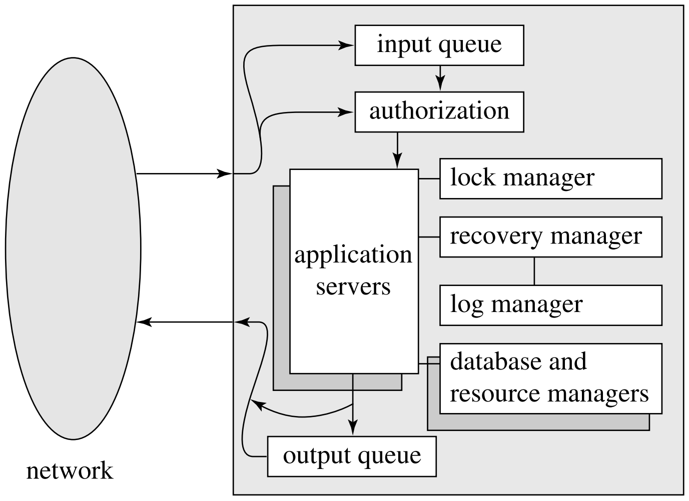


Figure 24.2 TP-monitor components.

durable queue helps ensure that once received and stored in the queue, the messages will be processed eventually, regardless of system failures. Authorization and application-server management (for example, server startup, and routing of messages to servers) are further functions of a TP monitor. TP monitors often provide logging, recovery, and concurrency-control facilities, allowing application servers to implement the ACID transaction properties directly if required.

Finally, TP monitors also provide support for persistent messaging. Recall that persistent messaging (Section 19.4.3) provides a guarantee that the message will be delivered if (and only if) the transaction commits.

In addition to these facilities, many TP monitors also provided *presentation facilities* to create menus/forms interfaces for dumb clients such as terminals; these facilities are no longer important since dumb clients are no longer widely used.

24.1.2 Application Coordination Using TP monitors

Applications today often have to interact with multiple databases. They may also have to interact with legacy systems, such as special-purpose data-storage systems built directly on file systems. Finally, they may have to communicate with users or other applications at remote sites. Hence, they also have to interact with communication subsystems. It is important to be able to coordinate data accesses, and to implement ACID properties for transactions across such systems.

Modern TP monitors provide support for the construction and administration of such large applications, built up from multiple subsystems such as databases, legacy systems, and communication systems. A TP monitor treats each subsystem as a **resource manager** that provides transactional access to some set of resources. The interface between the TP monitor and the resource manager is defined by a set of transaction primitives, such as *begin_transaction*, *commit_transaction*, *abort_transaction*, and *prepare_to_commit_transaction* (for two-phase commit). Of course, the resource manager must also provide other services, such as supplying data, to the application.

The resource-manager interface is defined by the X/Open Distributed Transaction Processing standard. Many database systems support the X/Open standards, and can act as resource managers. TP monitors—as well as other products, such as SQL systems, that support the X/Open standards—can connect to the resource managers.

In addition, services provided by a TP monitor, such as persistent messaging and durable queues, act as resource managers supporting transactions. The TP monitor can act as coordinator of two-phase commit for transactions that access these services as well as database systems. For example, when a queued update transaction is executed, an output message is delivered, and the request transaction is removed from the request queue. Two-phase commit between the database and the resource managers for the durable queue and persistent messaging helps ensure that, regardless of failures, either all these actions occur, or none occurs.

We can also use TP monitors to administer complex client–server systems consisting of multiple servers and a large number of clients. The TP monitor coordinates activities such as system checkpoints and shutdowns. It provides security and authentication of clients. It administers server pools by adding servers or removing servers without interruption of the system. Finally, it controls the scope of failures. If a server fails, the TP monitor can detect this failure, abort the transactions in progress, and restart the transactions. If a node fails, the TP monitor can migrate transactions to servers at other nodes, again backing out incomplete transactions. When failed nodes restart, the TP monitor can govern the recovery of the node's resource managers.

TP monitors can be used to hide database failures in replicated systems; remote backup systems (Section 17.10) are an example of replicated systems. Transaction requests are sent to the TP monitor, which relays the messages to one of the database replicas (the primary site, in case of remote backup systems). If one site fails, the TP monitor can transparently route messages to a backup site, masking the failure of the first site.

In client–server systems, clients often interact with servers via a **remote-procedure-call** (**RPC**) mechanism, where a client invokes a procedure call, which is actually executed at the server, with the results sent back to the client. As far as the client code that invokes the RPC is concerned, the call looks like a local procedure-call invocation. TP monitor systems, such as Encina, provide a **transactional RPC** interface to their services. In such an interface, the RPC mechanism provides calls that can be used to enclose a series of RPC calls within a transaction. Thus, updates performed by an RPC are carried out within the scope of the transaction, and can be rolled back if there is any failure.

24.2 Transactional Workflows

A **workflow** is an activity in which multiple tasks are executed in a coordinated way by different processing entities. A **task** defines some work to be done and can be specified in a number of ways, including a textual description in a file or electronic-mail message, a form, a message, or a computer program. The **processing entity** that

Workflow application	Typical task	Typical processing entity
electronic-mail routing	electronic-mail message	mailers
loan processing	form processing	humans, application software
purchase-order processing	form processing	humans, application software, DBMSs

Figure 24.3 Examples of workflows.

performs the tasks may be a person or a software system (for example, a mailer, an application program, or a database-management system).

Figure 24.3 shows examples of workflows. A simple example is that of an electronic-mail system. The delivery of a single mail message may involve several mailer systems that receive and forward the mail message, until the message reaches its destination, where it is stored. Each mailer performs a task—forwarding the mail to the next mailer—and the tasks of multiple mailers may be required to route mail from source to destination. Other terms used in the database and related literature to refer to workflows include **task flow** and **multisystem applications**. Workflow tasks are also sometimes called **steps**.

In general, workflows may involve one or more humans. For instance, consider the processing of a loan. The relevant workflow appears in Figure 24.4. The person who wants a loan fills out a form, which is then checked by a loan officer. An employee who processes loan applications verifies the data in the form, using sources such as credit-reference bureaus. When all the required information has been collected, the loan officer may decide to approve the loan; that decision may then have to be approved by one or more superior officers, after which the loan can be made. Each human here performs a task; in a bank that has not automated the task of loan processing, the coordination of the tasks is typically carried out by passing of the loan application, with attached notes and other information, from one employee to

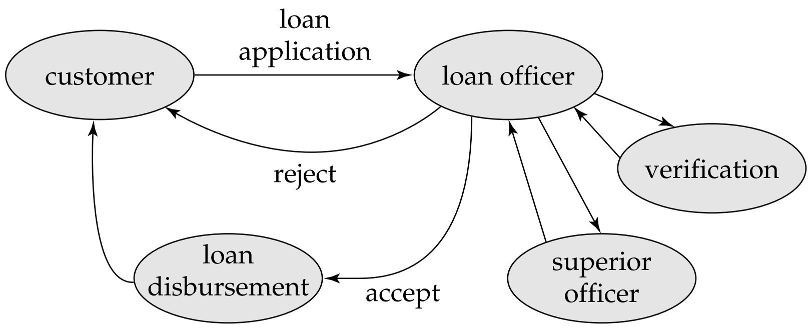


Figure 24.4 Workflow in loan processing.

the next. Other examples of workflows include processing of expense vouchers, of purchase orders, and of credit-card transactions.

Today, all the information related to a workflow is more than likely to be stored in a digital form on one or more computers, and, with the growth of networking, information can be easily transferred from one computer to another. Hence, it is feasible for organizations to automate their workflows. For example, to automate the tasks involved in loan processing, we can store the loan application and associated information in a database. The workflow itself then involves handing of responsibility from one human to the next, and possibly even to programs that can automatically fetch the required information. Humans can coordinate their activities by means such as electronic mail.

We have to address two activities, in general, to automate a workflow. The first is **workflow specification**: detailing the tasks that must be carried out and defining the execution requirements. The second problem is **workflow execution**, which we must do while providing the safeguards of traditional database systems related to computation correctness and data integrity and durability. For example, it is not acceptable for a loan application or a voucher to be lost, or to be processed more than once, because of a system crash. The idea behind transactional workflows is to use and extend the concepts of transactions to the context of workflows.

Both activities are complicated by the fact that many organizations use several independently managed information-processing systems that, in most cases, were developed separately to automate different functions. Workflow activities may require interactions among several such systems, each performing a task, as well as interactions with humans.

A number of workflow systems have been developed in recent years. Here, we study properties of workflow systems at a relatively abstract level, without going into the details of any particular system.

24.2.1 Workflow Specification

Internal aspects of a task do not need to be modeled for the purpose of specification and management of a workflow. In an abstract view of a task, a task may use parameters stored in its input variables, may retrieve and update data in the local system, may store its results in its output variables, and may be queried about its execution state. At any time during the execution, the **workflow state** consists of the collection of states of the workflow’s constituent tasks, and the states (values) of all variables in the workflow specification.

The coordination of tasks can be specified either statically or dynamically. A static specification defines the tasks—and dependencies among them—before the execution of the workflow begins. For instance, the tasks in an expense-voucher workflow may consist of the approvals of the voucher by a secretary, a manager, and an accountant, in that order, and finally by the delivery of a check. The dependencies among the tasks may be simple—each task has to be completed before the next begins.

A generalization of this strategy is to have a precondition for execution of each task in the workflow, so that all possible tasks in a workflow and their dependencies are known in advance, but only those tasks whose preconditions are satisfied are executed. The preconditions can be defined through dependencies such as the following:

- **Execution states** of other tasks—for example, "task t_i cannot start until task t_j has ended," or "task t_i must abort if task t_j has committed"
- **Output values** of other tasks—for example, "task t_i can start if task t_j returns a value greater than 25," or "the manager-approval task can start if the secretary-approval task returns a value of OK"
- **External variables** modified by external events—for example, "task t_i cannot be started before 9 AM," or "task t_i must be started within 24 hours of the completion of task t_j"

We can combine the dependencies by the regular logical connectors (**or**, **and**, **not**) to form complex scheduling preconditions.

An example of dynamic scheduling of tasks is an electronic-mail routing system. The next task to be scheduled for a given mail message depends on what the destination address of the message is, and on which intermediate routers are functioning.

24.2.2 Failure-Atomicity Requirements of a Workflow

The workflow designer may specify the **failure-atomicity** requirements of a workflow according to the semantics of the workflow. The traditional notion of failure atomicity would require that a failure of any task results in the failure of the workflow. However, a workflow can, in many cases, survive the failure of one of its tasks—for example, by executing a functionally equivalent task at another site. Therefore, we should allow the designer to define failure-atomicity requirements of a workflow. The system must guarantee that every execution of a workflow will terminate in a state that satisfies the failure-atomicity requirements defined by the designer. We call those states **acceptable termination states** of a workflow. All other execution states of a workflow constitute a set of **nonacceptable termination states**, in which the failure-atomicity requirements may be violated.

An acceptable termination state can be designated as committed or aborted. A **committed acceptable termination state** is an execution state in which the objectives of a workflow have been achieved. In contrast, an **aborted acceptable termination state** is a valid termination state in which a workflow has failed to achieve its objectives. If an aborted acceptable termination state has been reached, all undesirable effects of the partial execution of the workflow must be undone in accordance with that workflow's failure-atomicity requirements.

A workflow must reach an acceptable termination state *even in the presence of system failures*. Thus, if a workflow was in a nonacceptable termination state at the time of failure, during system recovery it must be brought to an acceptable termination state (whether aborted or committed).

For example, in the loan-processing workflow, in the final state, either the loan applicant is told that a loan cannot be made or the loan is disbursed. In case of failures such as a long failure of the verification system, the loan application could be returned to the loan applicant with a suitable explanation; this outcome would constitute an aborted acceptable termination. A committed acceptable termination would be either the acceptance or the rejection of the loan.

In general, a task can commit and release its resources before the workflow reaches a termination state. However, if the multitask transaction later aborts, its failure atomicity may require that we undo the effects of already completed tasks (for example, committed subtransactions) by executing compensating tasks (as subtransactions). The semantics of compensation requires that a compensating transaction eventually complete its execution successfully, possibly after a number of resubmissions.

In an expense-voucher-processing workflow, for example, a department-budget balance may be reduced on the basis of an initial approval of a voucher by the manager. If the voucher is later rejected, whether because of failure or for other reasons, the budget may have to be restored by a compensating transaction.

24.2.3 Execution of Workflows

The execution of the tasks may be controlled by a human coordinator or by a software system called a **workflow-management system**. A workflow-management system consists of a scheduler, task agents, and a mechanism to query the state of the workflow system. A task agent controls the execution of a task by a processing entity. A scheduler is a program that processes workflows by submitting various tasks for execution, monitoring various events, and evaluating conditions related to intertask dependencies. A scheduler may submit a task for execution (to a task agent), or may request that a previously submitted task be aborted. In the case of multidatabase transactions, the tasks are subtransactions, and the processing entities are local database management systems. In accordance with the workflow specifications, the scheduler enforces the scheduling dependencies and is responsible for ensuring that tasks reach acceptable termination states.

There are three architectural approaches to the development of a workflow-management system. A **centralized architecture** has a single scheduler that schedules the tasks for all concurrently executing workflows. The **partially distributed architecture** has one scheduler instantiated for each workflow. When the issues of concurrent execution can be separated from the scheduling function, the latter option is a natural choice. A **fully distributed architecture** has no scheduler, but the task agents coordinate their execution by communicating with one another to satisfy task dependencies and other workflow execution requirements.

The simplest workflow-execution systems follow the fully distributed approach just described and are based on messaging. Messaging may be implemented by persistent messaging mechanisms, to provide guaranteed delivery. Some implementations use e-mail for messaging; such implementations provide many of the features of persistent messaging, but generally do not guarantee atomicity of message delivery and transaction commit. Each site has a task agent that executes tasks received through messages. Execution may also involve presenting messages to humans, who have then to carry out some action. When a task is completed at a site, and needs to be processed at another site, the task agent dispatches a message to the next site. The message contains all relevant information about the task to be performed. Such message-based workflow systems are particularly useful in networks that may be disconnected for part of the time, such as dial-up networks.

The centralized approach is used in workflow systems where the data are stored in a central database. The scheduler notifies various agents, such as humans or computer programs, that a task has to be carried out, and keeps track of task completion. It is easier to keep track of the state of a workflow with a centralized approach than it is with a fully distributed approach.

The scheduler must guarantee that a workflow will terminate in one of the specified acceptable termination states. Ideally, before attempting to execute a workflow, the scheduler should examine that workflow to check whether the workflow may terminate in a nonacceptable state. If the scheduler cannot guarantee that a workflow will terminate in an acceptable state, it should reject such specifications without attempting to execute the workflow. As an example, let us consider a workflow consisting of two tasks represented by subtransactions S_1 and S_2, with the failure-atomicity requirements indicating that either both or neither of the subtransactions should be committed. If S_1 and S_2 do not provide prepared-to-commit states (for a two-phase commit), and further do not have compensating transactions, then it is possible to reach a state where one subtransaction is committed and the other aborted, and there is no way to bring both to the same state. Therefore, such a workflow specification is **unsafe**, and should be rejected.

Safety checks such as the one just described may be impossible or impractical to implement in the scheduler; it then becomes the responsibility of the person designing the workflow specification to ensure that the workflows are safe.

24.2.4 Recovery of a Workflow

The objective of **workflow recovery** is to enforce the failure atomicity of the workflows. The recovery procedures must make sure that, if a failure occurs in any of the workflow-processing components (including the scheduler), the workflow will eventually reach an acceptable termination state (whether aborted or committed). For example, the scheduler could continue processing after failure and recovery, as though nothing happened, thus providing forward recoverability. Otherwise, the scheduler could abort the whole workflow (that is, reach one of the global abort states). In either case, some subtransactions may need to be committed or even submitted for execution (for example, compensating subtransactions).

We assume that the processing entities involved in the workflow have their own local recovery systems and handle their local failures. To recover the execution-environment context, the failure-recovery routines need to restore the state information of the scheduler at the time of failure, including the information about the execution states of each task. Therefore, the appropriate status information must be logged on stable storage.

We also need to consider the contents of the message queues. When one agent hands off a task to another, the handoff should be carried out exactly once: If the handoff happens twice a task may get executed twice; if the handoff does not occur, the task may get lost. Persistent messaging (Section 19.4.3) provides exactly the features to ensure positive, single handoff.

24.2.5 Workflow Management Systems

Workflows are often hand coded as part of application systems. For instance, enterprise resource planning (ERP) systems, which help coordinate activities across an entire enterprise, have numerous workflows built into them.

The goal of workflow management systems is to simplify the construction of workflows and make them more reliable, by permitting them to be specified in a high-level manner and executed in accordance with the specification. There are a large number of commercial workflow management systems; some, like FlowMark from IBM, are general-purpose workflow management systems, while others are specific to particular workflows, such as order processing or bug/failure reporting systems.

In today's world of interconnected organizations, it is not sufficient to manage workflows only within an organization. Workflows that cross organizational boundaries are becoming increasingly common. For instance, consider an order placed by an organization and communicated to another organization that fulfills the order. In each organization there may be a workflow associated with the order, and it is important that the workflows be able to interoperate, in order to minimize human intervention.

The Workflow Management Coalition has developed standards for interoperation between workflow systems. Current standardization efforts use XML as the underlying language for communicating information about the workflow. See the bibliographical notes for more information.

24.3 Main-Memory Databases

To allow a high rate of transaction processing (hundreds or thousands of transactions per second), we must use high-performance hardware, and must exploit parallelism. These techniques alone, however, are insufficient to obtain very low response times, since disk I/O remains a bottleneck—about 10 milliseconds are required for each I/O and this number has not decreased at a rate comparable to the increase in processor speeds. Disk I/O is often the bottleneck for reads, as well as for transaction commits. The long disk latency (about 10 milliseconds average) increases not only the time to access a data item, but also limits the number of accesses per second.

We can make a database system less disk bound by increasing the size of the database buffer. Advances in main-memory technology let us construct large main memories at relatively low cost. Today, commercial 64-bit systems can support main memories of tens of gigabytes.

For some applications, such as real-time control, it is necessary to store data in main memory to meet performance requirements. The memory size required for most such systems is not exceptionally large, although there are at least a few applications that require multiple gigabytes of data to be memory resident. Since memory sizes have been growing at a very fast rate, an increasing number of applications can be expected to have data that fit into main memory.

Large main memories allow faster processing of transactions, since data are memory resident. However, there are still disk-related limitations:

- Log records must be written to stable storage before a transaction is committed. The improved performance made possible by a large main memory may result in the logging process becoming a bottleneck. We can reduce commit time by creating a stable log buffer in main memory, using nonvolatile RAM (implemented, for example, by battery backed-up memory). The overhead imposed by logging can also be reduced by the *group-commit* technique discussed later in this section. Throughput (number of transactions per second) is still limited by the data-transfer rate of the log disk.

- Buffer blocks marked as modified by committed transactions still have to be written so that the amount of log that has to be replayed at recovery time is reduced. If the update rate is extremely high, the disk data-transfer rate may become a bottleneck.

- If the system crashes, all of main memory is lost. On recovery, the system has an empty database buffer, and data items must be input from disk when they are accessed. Therefore, even after recovery is complete, it takes some time before the database is fully loaded in main memory and high-speed processing of transactions can resume.

On the other hand, a main-memory database provides opportunities for optimizations:

- Since memory is costlier than disk space, internal data structures in main-memory databases have to be designed to reduce space requirements. However, data structures can have pointers crossing multiple pages unlike those in disk databases, where the cost of the I/Os to traverse multiple pages would be excessively high. For example, tree structures in main-memory databases can be relatively deep, unlike B^+-trees, but should minimize space requirements.

- There is no need to pin buffer pages in memory before data are accessed, since buffer pages will never be replaced.

- Query-processing techniques should be designed to minimize space overhead, so that main memory limits are not exceeded while a query is being evaluated; that situation would result in paging to swap area, and would slow down query processing.

- Once the disk I/O bottleneck is removed, operations such as locking and latching may become bottlenecks. Such bottlenecks must be eliminated by improvements in the implementation of these operations.

- Recovery algorithms can be optimized, since pages rarely need to be written out to make space for other pages.

TimesTen and DataBlitz are two main-memory database products that exploit several of these optimizations, while the Oracle database has added special features to support very large main memories. Additional information on main-memory databases is given in the references in the bibliographical notes.

The process of committing a transaction T requires these records to be written to stable storage:

- All log records associated with T that have not been output to stable storage
- The <T **commit**> log record

These output operations frequently require the output of blocks that are only partially filled. To ensure that nearly full blocks are output, we use the **group-commit** technique. Instead of attempting to commit T when T completes, the system waits until several transactions have completed, or a certain period of time has passed since a transaction completed execution. It then commits the group of transactions that are waiting, together. Blocks written to the log on stable storage would contain records of several transactions. By careful choice of group size and maximum waiting time, the system can ensure that blocks are full when they are written to stable storage without making transactions wait excessively. This technique results, on average, in fewer output operations per committed transaction.

Although group commit reduces the overhead imposed by logging, it results in a slight delay in commit of transactions that perform updates. The delay can be made quite small (say, 10 milliseconds), which is acceptable for many applications. These delays can be eliminated if disks or disk controllers support nonvolatile RAM buffers for write operations. Transactions can commit as soon as the write is performed on the nonvolatile RAM buffer. In this case, there is no need for group commit.

Note that group commit is useful even in databases with disk-resident data.

24.4 Real-Time Transaction Systems

The integrity constraints that we have considered thus far pertain to the values stored in the database. In certain applications, the constraints include **deadlines** by which a task must be completed. Examples of such applications include plant management, traffic control, and scheduling. When deadlines are included, correctness of an execution is no longer solely an issue of database consistency. Rather, we are concerned with how many deadlines are missed, and by how much time they are missed. Deadlines are characterized as follows:

- **Hard deadline**. Serious problems, such as system crash, may occur if a task is not completed by its deadline.
- **Firm deadline**. The task has zero value if it is completed after the deadline.
- **Soft deadlines**. The task has diminishing value if it is completed after the deadline, with the value approaching zero as the degree of lateness increases.

Systems with deadlines are called **real-time systems**.

Transaction management in real-time systems must take deadlines into account. If the concurrency-control protocol determines that a transaction T_i must wait, it may cause T_i to miss the deadline. In such cases, it may be preferable to pre-empt the transaction holding the lock, and to allow T_i to proceed. Pre-emption must be used with care, however, because the time lost by the pre-empted transaction (due to rollback and restart) may cause the transaction to miss its deadline. Unfortunately, it is difficult to determine whether rollback or waiting is preferable in a given situation.

A major difficulty in supporting real-time constraints arises from the variance in transaction execution time. In the best case, all data accesses reference data in the database buffer. In the worst case, each access causes a buffer page to be written to disk (preceded by the requisite log records), followed by the reading from disk of the page containing the data to be accessed. Because the two or more disk accesses required in the worst case take several orders of magnitude more time than the main-memory references required in the best case, transaction execution time can be estimated only very poorly if data are resident on disk. Hence, main-memory databases are often used if real-time constraints have to be met.

However, even if data are resident in main memory, variances in execution time arise from lock waits, transaction aborts, and so on. Researchers have devoted considerable effort to concurrency control for real-time databases. They have extended locking protocols to provide higher priority for transactions with early deadlines. They have found that optimistic concurrency protocols perform well in real-time databases; that is, these protocols result in fewer missed deadlines than even the extended locking protocols. The bibliographical notes provide references to research in the area of real-time databases.

In real-time systems, deadlines, rather than absolute speed, are the most important issue. Designing a real-time system involves ensuring that there is enough processing power to meet deadlines without requiring excessive hardware resources. Achieving this objective, despite the variance in execution time resulting from transaction management, remains a challenging problem.

24.5 Long-Duration Transactions

The transaction concept developed initially in the context of data-processing applications, in which most transactions are noninteractive and of short duration. Although the techniques presented here and earlier in Chapters 15, 16, and 17 work well in those applications, serious problems arise when this concept is applied to database systems that involve human interaction. Such transactions have these key properties:

- **Long duration**. Once a human interacts with an active transaction, that transaction becomes a **long-duration transaction** from the perspective of the computer, since human response time is slow relative to computer speed. Furthermore, in design applications, the human activity may involve hours, days, or an even longer period. Thus, transactions may be of long duration in human terms, as well as in machine terms.

- **Exposure of uncommitted data**. Data generated and displayed to a user by a long-duration transaction are uncommitted, since the transaction may abort. Thus, users—and, as a result, other transactions—may be forced to read uncommitted data. If several users are cooperating on a project, user transactions may need to exchange data prior to transaction commit.

- **Subtasks**. An interactive transaction may consist of a set of subtasks initiated by the user. The user may wish to abort a subtask without necessarily causing the entire transaction to abort.
- **Recoverability**. It is unacceptable to abort a long-duration interactive transaction because of a system crash. The active transaction must be recovered to a state that existed shortly before the crash so that relatively little human work is lost.
- **Performance**. Good performance in an interactive transaction system is defined as fast response time. This definition is in contrast to that in a noninteractive system, in which high throughput (number of transactions per second) is the goal. Systems with high throughput make efficient use of system resources. However, in the case of interactive transactions, the most costly resource is the user. If the efficiency and satisfaction of the user is to be optimized, response time should be fast (from a human perspective). In those cases where a task takes a long time, response time should be predictable (that is, the variance in response times should be low), so that users can manage their time well.

In Sections 24.5.1 through 24.5.5, we shall see why these five properties are incompatible with the techniques presented thus far, and shall discuss how those techniques can be modified to accommodate long-duration interactive transactions.

24.5.1 Nonserializable Executions

The properties that we discussed make it impractical to enforce the requirement used in earlier chapters that only serializable schedules be permitted. Each of the concurrency-control protocols of Chapter 16 has adverse effects on long-duration transactions:

- **Two-phase locking**. When a lock cannot be granted, the transaction requesting the lock is forced to wait for the data item in question to be unlocked. The duration of this wait is proportional to the duration of the transaction holding the lock. If the data item is locked by a short-duration transaction, we expect that the waiting time will be short (except in case of deadlock or extraordinary system load). However, if the data item is locked by a long-duration transaction, the wait will be of long duration. Long waiting times lead to both longer response time and an increased chance of deadlock.
- **Graph-based protocols**. Graph-based protocols allow for locks to be released earlier than under the two-phase locking protocols, and they prevent deadlock. However, they impose an ordering on the data items. Transactions must lock data items in a manner consistent with this ordering. As a result, a transaction may have to lock more data than it needs. Furthermore, a transaction must hold a lock until there is no chance that the lock will be needed again. Thus, long-duration lock waits are likely to occur.

- **Timestamp-based protocols**. Timestamp protocols never require a transaction to wait. However, they do require transactions to abort under certain circumstances. If a long-duration transaction is aborted, a substantial amount of work is lost. For noninteractive transactions, this lost work is a performance issue. For interactive transactions, the issue is also one of user satisfaction. It is highly undesirable for a user to find that several hours' worth of work have been undone.

- **Validation protocols**. Like timestamp-based protocols, validation protocols enforce serializability by means of transaction abort.

Thus, it appears that the enforcement of serializability results in long-duration waits, in abort of long-duration transactions, or in both. There are theoretical results, cited in the bibliographical notes, that substantiate this conclusion.

Further difficulties with the enforcement of serializability arise when we consider recovery issues. We previously discussed the problem of cascading rollback, in which the abort of a transaction may lead to the abort of other transactions. This phenomenon is undesirable, particularly for long-duration transactions. If locking is used, exclusive locks must be held until the end of the transaction, if cascading rollback is to be avoided. This holding of exclusive locks, however, increases the length of transaction waiting time.

Thus, it appears that the enforcement of transaction atomicity must either lead to an increased probability of long-duration waits or create a possibility of cascading rollback.

These considerations are the basis for the alternative concepts of correctness of concurrent executions and transaction recovery that we consider in the remainder of this section.

24.5.2 Concurrency Control

The fundamental goal of database concurrency control is to ensure that concurrent execution of transactions does not result in a loss of database consistency. The concept of serializability can be used to achieve this goal, since all serializable schedules preserve consistency of the database. However, not all schedules that preserve consistency of the database are serializable. For an example, consider again a bank database consisting of two accounts A and B, with the consistency requirement that the sum $A + B$ be preserved. Although the schedule of Figure 24.5 is not conflict serializable, it nevertheless preserves the sum of $A + B$. It also illustrates two important points about the concept of correctness without serializability.

- Correctness depends on the specific consistency constraints for the database.
- Correctness depends on the properties of operations performed by each transaction.

In general it is not possible to perform an automatic analysis of low-level operations by transactions and check their effect on database consistency constraints. However, there are simpler techniques. One is to use the database consistency constraints as

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Figure 24.5 A non-conflict-serializable schedule.

the basis for a split of the database into subdatabases on which concurrency can be managed separately. Another is to treat some operations besides **read** and **write** as fundamental low-level operations, and to extend concurrency control to deal with them.

The bibliographical notes reference other techniques for ensuring consistency without requiring serializability. Many of these techniques exploit variants of multiversion concurrency control (see Section 17.6). For older data-processing applications that need only one version, multiversion protocols impose a high space overhead to store the extra versions. Since many of the new database applications require the maintenance of versions of data, concurrency-control techniques that exploit multiple versions are practical.

24.5.3 Nested and Multilevel Transactions

A long-duration transaction can be viewed as a collection of related subtasks or subtransactions. By structuring a transaction as a set of subtransactions, we are able to enhance parallelism, since it may be possible to run several subtransactions in parallel. Furthermore, it is possible to deal with failure of a subtransaction (due to abort, system crash, and so on) without having to roll back the entire long-duration transaction.

A nested or multilevel transaction T consists of a set $T = \{t_1, t_2, \ldots, t_n\}$ of subtransactions and a partial order P on T. A subtransaction t_i in T may abort without forcing T to abort. Instead, T may either restart t_i or simply choose not to run t_i. If t_i commits, this action does not make t_i permanent (unlike the situation in Chapter 17). Instead, t_i *commits to* T, and may still abort (or require compensation—see Section 24.5.4) if T aborts. An execution of T must not violate the partial order P. That is, if an edge $t_i \rightarrow t_j$ appears in the precedence graph, then $t_j \rightarrow t_i$ must not be in the transitive closure of P.

Nesting may be several levels deep, representing a subdivision of a transaction into subtasks, subsubtasks, and so on. At the lowest level of nesting, we have the standard database operations **read** and **write** that we have used previously.

If a subtransaction of T is permitted to release locks on completion, T is called a **multilevel transaction**. When a multilevel transaction represents a long-duration activity, the transaction is sometimes referred to as a **saga**. Alternatively, if locks held by a subtransaction t_i of T are automatically assigned to T on completion of t_i, T is called a **nested transaction**.

Although the main practical value of multilevel transactions arises in complex, long-duration transactions, we shall use the simple example of Figure 24.5 to show how nesting can create higher-level operations that may enhance concurrency. We rewrite transaction T_1, using subtransactions $T_{1,1}$ and $T_{1,2}$, which perform increment or decrement operations:

- T_1 consists of
  - ☐ $T_{1,1}$, which subtracts 50 from *A*
  - ☐ $T_{1,2}$, which adds 50 to *B*

Similarly, we rewrite transaction T_2, using subtransactions $T_{2,1}$ and $T_{2,2}$, which also perform increment or decrement operations:

- T_2 consists of
  - ☐ $T_{2,1}$, which subtracts 10 from *B*
  - ☐ $T_{2,2}$, which adds 10 to *A*

No ordering is specified on $T_{1,1},\ T_{1,2},\ T_{2,1}$, and $T_{2,2}$. Any execution of these subtransactions will generate a correct result. The schedule of Figure 24.5 corresponds to the schedule $< T_{1,1},\ T_{2,1},\ T_{1,2},\ T_{2,2} >$.

24.5.4 Compensating Transactions

To reduce the frequency of long-duration waiting, we arrange for uncommitted updates to be exposed to other concurrently executing transactions. Indeed, multilevel transactions may allow this exposure. However, the exposure of uncommitted data creates the potential for cascading rollbacks. The concept of **compensating transactions** helps us to deal with this problem.

Let transaction T be divided into several subtransactions $t_1, t_2, \ldots, t_n$. After a subtransaction t_i commits, it releases its locks. Now, if the outer-level transaction T has to be aborted, the effect of its subtransactions must be undone. Suppose that subtransactions $t_1, \ldots, t_k$ have committed, and that t_{k+1} was executing when the decision to abort is made. We can undo the effects of t_{k+1} by aborting that subtransaction. However, it is not possible to abort subtransactions $t_1, \ldots, t_k$, since they have committed already.

Instead, we execute a new subtransaction ct_i, called a *compensating transaction*, to undo the effect of a subtransaction t_i. Each subtransaction t_i is required to have a compensating transaction ct_i. The compensating transactions must be executed in the inverse order $ct_k, \ldots, ct_1$. Here are several examples of compensation:

- Consider the schedule of Figure 24.5, which we have shown to be correct, although not conflict serializable. Each subtransaction releases its locks once it completes. Suppose that T_2 fails just prior to termination, after $T_{2,2}$ has released its locks. We then run a compensating transaction for $T_{2,2}$ that subtracts 10 from A and a compensating transaction for $T_{2,1}$ that adds 10 to B.

- Consider a database insert by transaction T_i that, as a side effect, causes a B$^+$-tree index to be updated. The insert operation may have modified several nodes of the B$^+$-tree index. Other transactions may have read these nodes in accessing data other than the record inserted by T_i. As in Section 17.9, we can undo the insertion by deleting the record inserted by T_i. The result is a correct, consistent B$^+$-tree, but is not necessarily one with exactly the same structure as the one we had before T_i started. Thus, deletion is a compensating action for insertion.

- Consider a long-duration transaction T_i representing a travel reservation. Transaction T has three subtransactions: $T_{i,1}$, which makes airline reservations; $T_{i,2}$, which reserves rental cars; and $T_{i,3}$, which reserves a hotel room. Suppose that the hotel cancels the reservation. Instead of undoing all of T_i, we compensate for the failure of $T_{i,3}$ by deleting the old hotel reservation and making a new one.

If the system crashes in the middle of executing an outer-level transaction, its subtransactions must be rolled back when it recovers. The techniques described in Section 17.9 can be used for this purpose.

Compensation for the failure of a transaction requires that the semantics of the failed transaction be used. For certain operations, such as incrementation or insertion into a B$^+$-tree, the corresponding compensation is easily defined. For more complex transactions, the application programmers may have to define the correct form of compensation at the time that the transaction is coded. For complex interactive transactions, it may be necessary for the system to interact with the user to determine the proper form of compensation.

24.5.5 Implementation Issues

The transaction concepts discussed in this section create serious difficulties for implementation. We present a few of them here, and discuss how we can address these problems.

Long-duration transactions must survive system crashes. We can ensure that they will by performing a **redo** on committed subtransactions, and by performing either an **undo** or compensation for any short-duration subtransactions that were active at the time of the crash. However, these actions solve only part of the problem. In typical database systems, such internal system data as lock tables and transactions timestamps are kept in volatile storage. For a long-duration transaction to be resumed after a crash, these data must be restored. Therefore, it is necessary to log not only changes to the database, but also changes to internal system data pertaining to long-duration transactions.

Logging of updates is made more complex when certain types of data items exist in the database. A data item may be a CAD design, text of a document, or another form of composite design. Such data items are physically large. Thus, storing both the old and new values of the data item in a log record is undesirable.

There are two approaches to reducing the overhead of ensuring the recoverability of large data items:

- **Operation logging**. Only the operation performed on the data item and the data-item name are stored in the log. Operation logging is also called **logical logging**. For each operation, an inverse operation must exist. We perform **undo** using the inverse operation, and **redo** using the operation itself. Recovery through operation logging is more difficult, since **redo** and **undo** are not idempotent. Further, using logical logging for an operation that updates multiple pages is greatly complicated by the fact that some, but not all, of the updated pages may have been written to the disk, so it is hard to apply either the **redo** or the **undo** of the operation on the disk image during recovery.

  Using physical redo logging and logical undo logging, as described in Section 17.9, provides the concurrency benefits of logical logging while avoiding the above pitfalls.

- **Logging and shadow paging**. Logging is used for modifications to small data items, but large data items are made recoverable via a shadow-page technique (see Section 17.5). When we use shadowing, only those pages that are actually modified need to be stored in duplicate.

Regardless of the technique used, the complexities introduced by long-duration transactions and large data items complicate the recovery process. Thus, it is desirable to allow certain noncritical data to be exempt from logging, and to rely instead on offline backups and human intervention.

24.6 Transaction Management in Multidatabases

Recall from Section 19.8 that a multidatabase system creates the illusion of logical database integration, in a heterogeneous database system where the local database systems may employ different logical data models and data-definition and data-manipulation languages, and may differ in their concurrency-control and transaction-management mechanisms.

A multidatabase system supports two types of transactions:

1. **Local transactions**. These transactions are executed by each local database system outside of the multidatabase system's control.

2. **Global transactions**. These transactions are executed under the multidatabase system's control.

The multidatabase system is aware of the fact that local transactions may run at the local sites, but it is not aware of what specific transactions are being executed, or of what data they may access.

Ensuring the local autonomy of each database system requires that no changes be made to its software. A database system at one site thus is not able to communicate directly with a one at any other site to synchronize the execution of a global transaction active at several sites.

Since the multidatabase system has no control over the execution of local transactions, each local system must use a concurrency-control scheme (for example, two-phase locking or timestamping) to ensure that its schedule is serializable. In addition, in case of locking, the local system must be able to guard against the possibility of local deadlocks.

The guarantee of local serializability is not sufficient to ensure global serializability. As an illustration, consider two global transactions T_1 and T_2, each of which accesses and updates two data items, A and B, located at sites S_1 and S_2, respectively. Suppose that the local schedules are serializable. It is still possible to have a situation where, at site S_1, T_2 follows T_1, whereas, at S_2, T_1 follows T_2, resulting in a nonserializable global schedule. Indeed, even if there is no concurrency among global transactions (that is, a global transaction is submitted only after the previous one commits or aborts), local serializability is not sufficient to ensure global serializability (see Exercise 24.14).

Depending on the implementation of the local database systems, a global transaction may not be able to control the precise locking behavior of its local substransactions. Thus, even if all local database systems follow two-phase locking, it may be possible only to ensure that each local transaction follows the rules of the protocol. For example, one local database system may commit its subtransaction and release locks, while the subtransaction at another local system is still executing. If the local systems permit control of locking behavior and all systems follow two-phase locking, then the multidatabase system can ensure that global transactions lock in a two-phase manner and the lock points of conflicting transactions would then define their global serialization order. If different local systems follow different concurrency-control mechanisms, however, this straightforward sort of global control does not work.

There are many protocols for ensuring consistency despite concurrent execution of global and local transactions in multidatabase systems. Some are based on imposing sufficient conditions to ensure global serializability. Others ensure only a form of consistency weaker than serializability, but achieve this consistency by less restrictive means. We consider one of the latter schemes: *two-level serializability*. Section 24.5 describes further approaches to consistency without serializability; other approaches are cited in the bibliographical notes.

A related problem in multidatabase systems is that of global atomic commit. If all local systems follow the two-phase commit protocol, that protocol can be used to achieve global atomicity. However, local systems not designed to be part of a distributed system may not be able to participate in such a protocol. Even if a local system is capable of supporting two-phase commit, the organization owning the system may be unwilling to permit waiting in cases where blocking occurs. In such cases, compromises may be made that allow for lack of atomicity in certain failure modes. Further discussion of these matters appears in the literature (see the bibliographical notes).

24.6.1 Two-Level Serializability

Two-level serializability (**2LSR**) ensures serializability at two levels of the system:

- Each local database system ensures local serializability among its local transactions, including those that are part of a global transaction.
- The multidatabase system ensures serializability among the global transactions alone—*ignoring the orderings induced by local transactions*.

Each of these serializability levels is simple to enforce. Local systems already offer guarantees of serializability; thus, the first requirement is easy to achieve. The second requirement applies to only a projection of the global schedule in which local transactions do not appear. Thus, the multidatabase system can ensure the second requirement using standard concurrency-control techniques (the precise choice of technique does not matter).

The two requirements of 2LSR are not sufficient to ensure global serializability. However, under the 2LSR-based approach, we adopt a requirement weaker than serializability, called **strong correctness**:

1. Preservation of consistency as specified by a set of consistency constraints
2. Guarantee that the set of data items read by each transaction is consistent

It can be shown that certain restrictions on transaction behavior, combined with 2LSR, are sufficient to ensure strong correctness (although not necessarily to ensure serializability). We list several of these restrictions.

In each of the protocols, we distinguish between **local data** and **global data**. Local data items belong to a particular site and are under the sole control of that site. Note that there cannot be any consistency constraints between local data items at distinct sites. Global data items belong to the multidatabase system, and, though they may be stored at a local site, are under the control of the multidatabase system.

The **global-read protocol** allows global transactions to read, but not to update, local data items, while disallowing all access to global data by local transactions. The global-read protocol ensures strong correctness if all these conditions hold:

1. Local transactions access only local data items.
2. Global transactions may access global data items, and may read local data items (although they must not write local data items).
3. There are no consistency constraints between local and global data items.

The **local-read protocol** grants local transactions read access to global data, but disallows all access to local data by global transactions. In this protocol, we need to introduce the notion of a **value dependency**. A transaction has a value dependency if the value that it writes to a data item at one site depends on a value that it read for a data item on another site.

The local-read protocol ensures strong correctness if all these conditions hold:

1. Local transactions may access local data items, and may read global data items stored at the site (although they must not write global data items).
2. Global transactions access only global data items.
3. No transaction may have a value dependency.

The **global-read–write/local-read protocol** is the most generous in terms of data access of the protocols that we have considered. It allows global transactions to read and write local data, and allows local transactions to read global data. However, it imposes both the value-dependency condition of the local-read protocol and the condition from the global-read protocol that there be no consistency constraints between local and global data.

The global-read–write/local-read protocol ensures strong correctness if all these conditions hold:

1. Local transactions may access local data items, and may read global data items stored at the site (although they must not write global data items).
2. Global transactions may access global data items as well as local data items (that is, they may read and write all data).
3. There are no consistency constraints between local and global data items.
4. No transaction may have a value dependency.

24.6.2 Ensuring Global Serializability

Early multidatabase systems restricted global transactions to be read only. They thus avoided the possibility of global transactions introducing inconsistency to the data, but were not sufficiently restrictive to ensure global serializability. It is indeed possible to get such global schedules and to develop a scheme to ensure global serializability, and we ask you to do both in Exercise 24.15.

There are a number of general schemes to ensure global serializability in an environment where update as well read-only transactions can execute. Several of these schemes are based on the idea of a **ticket**. A special data item called a ticket is created in each local database system. Every global transaction that accesses data at a site must write the ticket at that site. This requirement ensures that global transactions conflict directly at every site they visit. Furthermore, the global transaction manager can control the order in which global transactions are serialized, by controlling the order in which the tickets are accessed. References to such schemes appear in the bibliographical notes.

If we want to ensure global serializability in an environment where no direct local conflicts are generated in each site, some assumptions must be made about the schedules allowed by the local database system. For example, if the local schedules are such that the commit order and serialization order are always identical, we can ensure serializability by controlling only the order in which transactions commit.

The problem with schemes that ensure global serializability is that they may restrict concurrency unduly. They are particularly likely to do so because most transactions submit SQL statements to the underlying database system, instead of submitting individual **read**, **write**, **commit**, and **abort** steps. Although it is still possible to ensure global serializability under this assumption, the level of concurrency may be such that other schemes, such as the two-level serializability technique discussed in Section 24.6.1, are attractive alternatives.

24.7 Summary

- Workflows are activities that involve the coordinated execution of multiple tasks performed by different processing entities. They exist not just in computer applications, but also in almost all organizational activities. With the growth of networks, and the existence of multiple autonomous database systems, workflows provide a convenient way of carrying out tasks that involve multiple systems.

- Although the usual ACID transactional requirements are too strong or are unimplementable for such workflow applications, workflows must satisfy a limited set of transactional properties that guarantee that a process is not left in an inconsistent state.

- Transaction-processing monitors were initially developed as multithreaded servers that could service large numbers of terminals from a single process. They have since evolved, and today they provide the infrastructure for building and administering complex transaction-processing systems that have a large number of clients and multiple servers. They provide services such as durable queueing of client requests and server responses, routing of client messages to servers, persistent messaging, load balancing, and coordination of two-phase commit when transactions access multiple servers.

- Large main memories are exploited in certain systems to achieve high system throughput. In such systems, logging is a bottleneck. Under the group-commit concept, the number of outputs to stable storage can be reduced, thus releasing this bottleneck.

- The efficient management of long-duration interactive transactions is more complex, because of the long-duration waits, and because of the possibility of aborts. Since the concurrency-control techniques used in Chapter 16 use waits, aborts, or both, alternative techniques must be considered. These techniques must ensure correctness without requiring serializability.

- A long-duration transaction is represented as a nested transaction with atomic database operations at the lowest level. If a transaction fails, only active short-duration transactions abort. Active long-duration transactions resume once any short-duration transactions have recovered. A compensating transaction is needed to undo updates of nested transactions that have committed, if the outer-level transaction fails.

- In systems with real-time constraints, correctness of execution involves not only database consistency but also deadline satisfaction. The wide variance of execution times for read and write operations complicates the transaction-management problem for time-constrained systems.
- A multidatabase system provides an environment in which new database applications can access data from a variety of pre-existing databases located in various heterogeneous hardware and software environments.

  The local database systems may employ different logical models and data-definition and data-manipulation languages, and may differ in their concurrency-control and transaction-management mechanisms. A multidatabase system creates the illusion of logical database integration, without requiring physical database integration.

Review Terms

- TP monitor
- TP-monitor architectures
  - ☐ Process per client
  - ☐ Single server
  - ☐ Many server, single router
  - ☐ Many server, many router
- Multitasking
- Context switch
- Multithreaded server
- Queue manager
- Application coordination
  - ☐ Resource manager
  - ☐ Remote procedure call (RPC)
- Transactional Workflows
  - ☐ Task
  - ☐ Processing entity
  - ☐ Workflow specification
  - ☐ Workflow execution
- Workflow state
  - ☐ Execution states
  - ☐ Output values
  - ☐ External variables
- Workflow failure atomicity
- Workflow termination states
  - ☐ Acceptable
  - ☐ Nonacceptable
  - ☐ Committed
  - ☐ Aborted
- Workflow recovery
- Workflow-management system
- Workflow-management system architectures
  - ☐ Centralized
  - ☐ Partially distributed
  - ☐ Fully distributed
- Main-memory databases
- Group commit
- Real-time systems
- Deadlines
  - ☐ Hard deadline
  - ☐ Firm deadline
  - ☐ Soft deadline
- Real-time databases
- Long-duration transactions
- Exposure of uncommitted data
- Subtasks

- Nonserializable executions
- Nested transactions
- Multilevel transactions
- Saga
- Compensating transactions
- Logical logging
- Multidatabase systems
- Autonomy
- Local transactions
- Global transactions
- Two-level serializability (2LSR)
- Strong correctness
- Local data
- Global data
- Protocols
  - ☐ Global-read
  - ☐ Local-read
  - ☐ Value dependency
  - ☐ Global-read–write/local-read
- Ensuring global serializability
- Ticket

Exercises

24.1 Explain how a TP monitor manages memory and processor resources more effectively than a typical operating system.

24.2 Compare TP monitor features with those provided by Web servers supporting servlets (such servers have been nicknamed *TP-lite*).

24.3 Consider the process of admitting new students at your university (or new employees at your organization).

a. Give a high-level picture of the workflow starting from the student application procedure.
b. Indicate acceptable termination states, and which steps involve human intervention.
c. Indicate possible errors (including deadline expiry) and how they are dealt with.
d. Study how much of the workflow has been automated at your university.

24.4 Like database systems, workflow systems also require concurrency and recovery management. List three reasons why we cannot simply apply a relational database system using 2PL, physical undo logging, and 2PC.

24.5 If the entire database fits in main memory, do we still need a database system to manage the data? Explain your answer.

24.6 Consider a main-memory database system recovering from a system crash. Explain the relative merits of

- Loading the entire database back into main memory before resuming transaction processing
- Loading data as it is requested by transactions

24.7 In the group-commit technique, how many transactions should be part of a group? Explain your answer.

24.8 Is a high-performance transaction system necessarily a real-time system? Why or why not?

24.9 In a database system using write-ahead logging, what is the worst-case number of disk accesses required to read a data item? Explain why this presents a problem to designers of real-time database systems.

24.10 Explain why it may be impractical to require serializability for long-duration transactions.

24.11 Consider a multithreaded process that delivers messages from a durable queue of persistent messages. Different threads may run concurrently, attempting to deliver different messages. In case of a delivery failure, the message must be restored in the queue. Model the actions that each thread carries out as a multilevel transaction, so that locks on the queue need not be held till a message is delivered.

24.12 Discuss the modifications that need to be made in each of the recovery schemes covered in Chapter 17 if we allow nested transactions. Also, explain any differences that result if we allow multilevel transactions.

24.13 What is the purpose of compensating transactions? Present two examples of their use.

24.14 Consider a multidatabase system in which it is guaranteed that at most one global transaction is active at any time, and every local site ensures local serializability.

a. Suggest ways in which the multidatabase system can ensure that there is at most one active global transaction at any time.
b. Show by example that it is possible for a nonserializable global schedule to result despite the assumptions.

24.15 Consider a multidatabase system in which every local site ensures local serializability, and all global transactions are read only.

a. Show by example that nonserializable executions may result in such a system.
b. Show how you could use a ticket scheme to ensure global serializability.

Bibliographical Notes

Gray and Edwards [1995] provides an overview of TP monitor architectures; Gray and Reuter [1993] provides a detailed (and excellent) textbook description of transaction-processing systems, including chapters on TP monitors. Our description of TP monitors is modeled on these two sources. X/Open [1991] defines the X/Open XA interface. Transaction processing in Tuxedo is described in Huffman [1993]. Wipfler [1987] is one of several texts on application development using CICS.

Fischer [2001] is a handbook on workflow systems. A reference model for workflows, proposed by the Workflow Management Coalition, is presented in Hollinsworth

[1994]. The Web site of the coalition is www.wfmc.org. Our description of workflows follows the model of Rusinkiewicz and Sheth [1995].

Reuter [1989] presents ConTracts, a method for grouping transactions into multitransaction activities. Some issues related to workflows were addressed in the work on long-running activities described by Dayal et al. [1990] and Dayal et al. [1991]. The authors propose event–condition–action rules as a technique for specifying workflows. Jin et al. [1993] describes workflow issues in telecommunication applications.

Garcia-Molina and Salem [1992] provides an overview of main-memory databases. Jagadish et al. [1993] describes a recovery algorithm designed for main-memory databases. A storage manager for main-memory databases is described in Jagadish et al. [1994].

Transaction processing in real-time databases is discussed by Abbott and Garcia-Molina [1999] and Dayal et al. [1990]. Barclay et al. [1982] describes a real-time database system used in a telecommunications switching system. Complexity and correctness issues in real-time databases are addressed by Korth et al. [1990b] and Soparkar et al. [1995]. Concurrency control and scheduling in real-time databases are discussed by Haritsa et al. [1990], Hong et al. [1993], and Pang et al. [1995]. Ozsoyoglu and Snodgrass [1995] is a survey of research in real-time and temporal databases.

Nested and multilevel transactions are presented by Lynch [1983], Moss [1982], Moss [1985], Lynch and Merritt [1986], Fekete et al. [1990b], Fekete et al. [1990a], Korth and Speegle [1994], and Pu et al. [1988]. Theoretical aspects of multilevel transactions are presented in Lynch et al. [1988] and Weihl and Liskov [1990].

Several extended-transaction models have been defined including Sagas (Garcia-Molina and Salem [1987]), ACTA (Chrysanthis and Ramamritham [1994]), the ConTract model (Wachter and Reuter [1992]), ARIES (Mohan et al. [1992] and Rothermel and Mohan [1989]), and the NT/PV model (Korth and Speegle [1994]).

Splitting transactions to achieve higher performance is addressed in Shasha et al. [1995]. A model for concurrency in nested transactions systems is presented in Beeri et al. [1989]. Relaxation of serializability is discussed in Garcia-Molina [1983] and Sha et al. [1988]. Recovery in nested transaction systems is discussed by Moss [1987], Haerder and Rothermel [1987], Rothermel and Mohan [1989]. Multilevel transaction management is discussed in Weikum [1991].

Gray [1981], Skarra and Zdonik [1989], Korth and Speegle [1988], and Korth and Speegle [1990] discuss long-duration transactions. Transaction processing for long-duration transactions is considered by Weikum and Schek [1984], Haerder and Rothermel [1987], Weikum et al. [1990], and Korth et al. [1990a]. Salem et al. [1994] presents an extension of 2PL for long-duration transactions by allowing the early release of locks under certain circumstances. Transaction processing in design and software-engineering applications is discussed in Korth et al. [1988], Kaiser [1990], and Weikum [1991].

Transaction processing in multidatabase systems is discussed in Breitbart et al. [1990], Breitbart et al. [1991], Breitbart et al. [1992], Soparkar et al. [1991], Mehrotra et al. [1992b] and Mehrotra et al. [1992a]. The ticket scheme is presented in Georgakopoulos et al. [1994]. 2LSR is introduced in Mehrotra et al. [1991]. An earlier approach, called *quasi-serializability*, is presented in Du and Elmagarmid [1989].