CHAPTER 23

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-

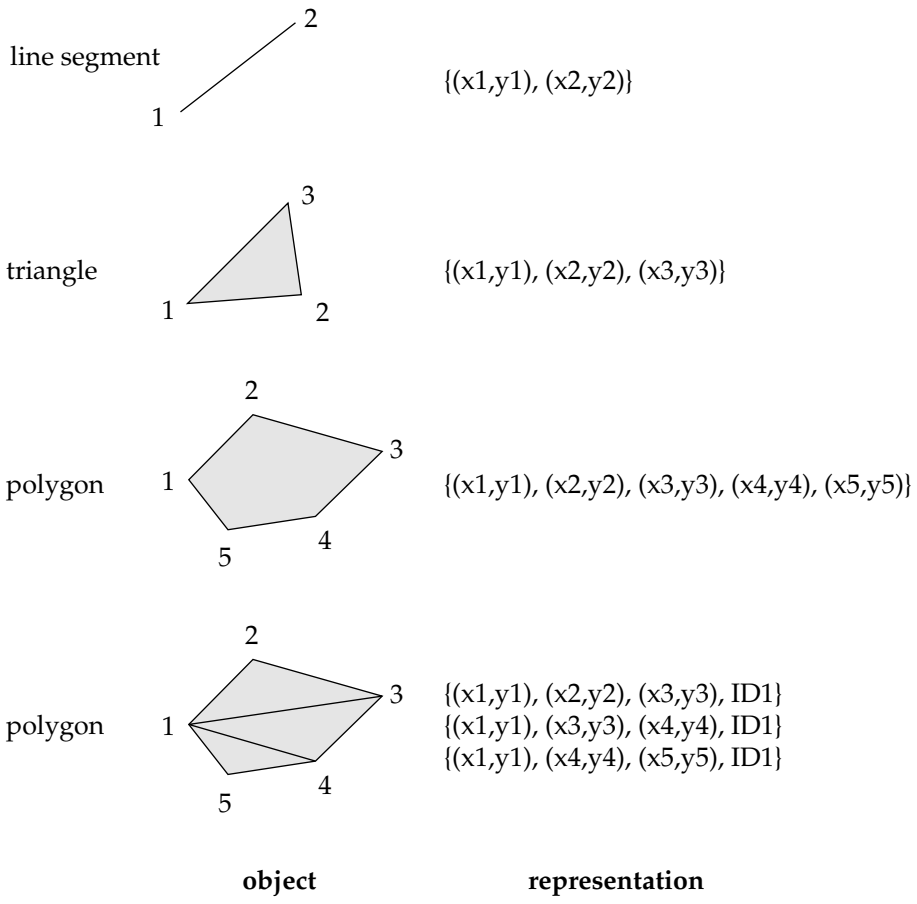


Figure 23.2 Representation of geometric constructs.

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.

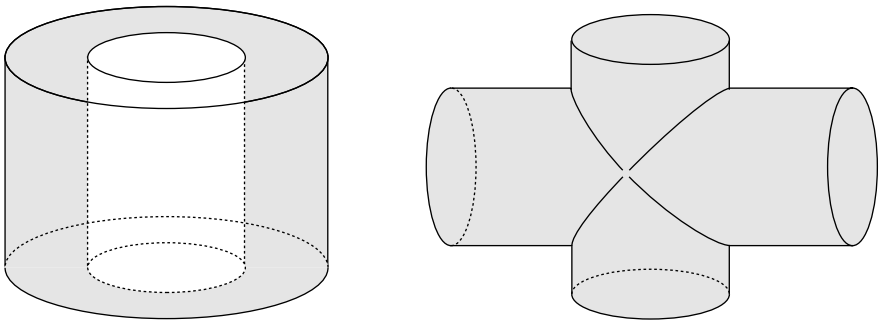
systems. Object-oriented databases represent components of the design as objects, and the connections between the objects indicate how the design is structured.

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.

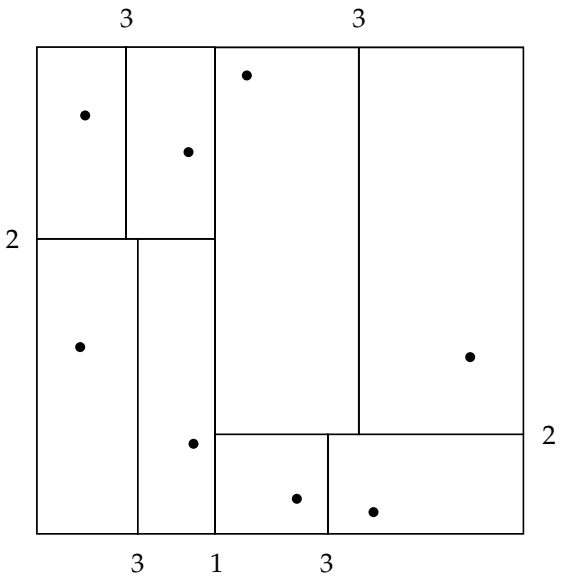


Figure 23.4 Division of space by a k-d tree.

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 nonleaf 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.

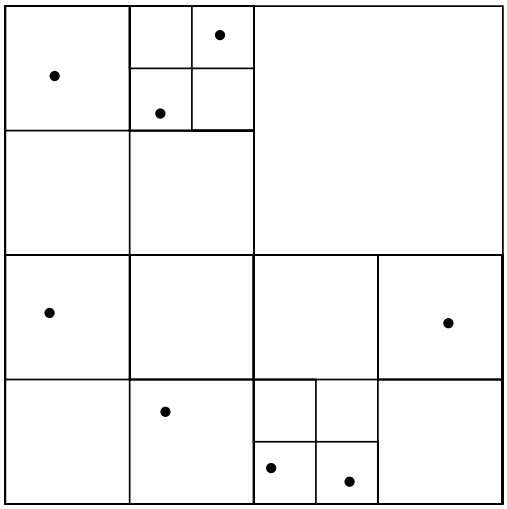


Figure 23.5 Division of space by a quadtree.

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.

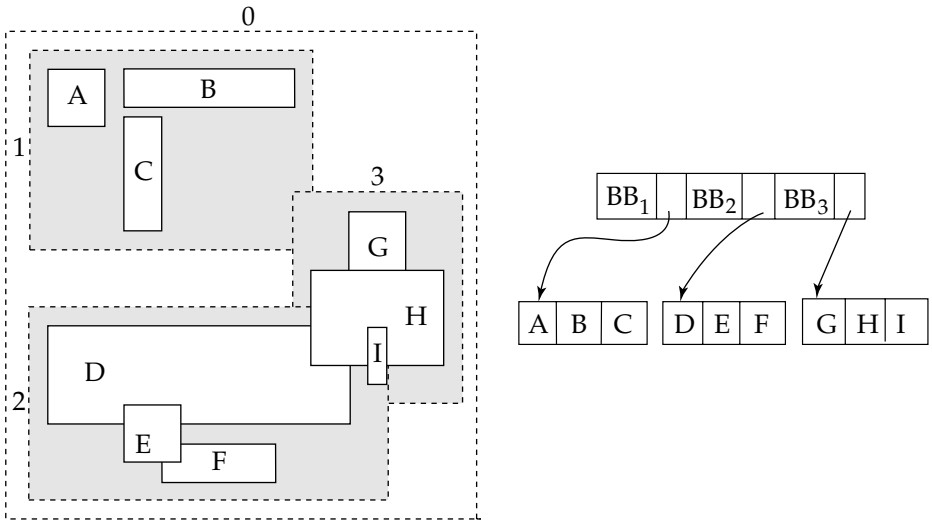


Figure 23.6 An R-tree.

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 up-

dating 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].