26 SPATIAL DATA MANAGEMENT

> Nothing puzzles me more than time and space; and yet nothing puzzles me less, as I never think about them.
>
> —Charles Lamb

Many applications involve large collections of spatial objects, and querying, indexing and maintaining such collections requires some specialized techniques. In this chapter, we motivate spatial data management and provide an introduction to the required techniques.

We introduce the different kinds of spatial data and queries in Section 26.1 and discuss several motivating applications in Section 26.2. We explain why indexing structures such as B+ trees are not adequate for handling spatial data in Section 26.3. We discuss three approaches to indexing spatial data in Sections 26.4 through 26.6. In Section 26.4, we discuss indexing techniques based on space-filling curves; in Section 26.5, we discuss the Grid file, an indexing technique that partitions the data space into nonoverlapping regions; and in Section 26.6, we discuss the R tree, an indexing technique based upon hierarchical partitioning of the data space into possibly overlapping regions. Finally, in Section 26.7 we discuss some issues that arise in indexing datasets with a large number of dimensions.

26.1 TYPES OF SPATIAL DATA AND QUERIES

We use the term **spatial data** in a broad sense, covering multidimensional points, lines, rectangles, polygons, cubes, and other geometric objects. A spatial data object occupies a certain region of space, called its **spatial extent**, which is characterized by its **location** and **boundary**.

From the point of view of a DBMS, we can classify spatial data as being either *point data* or *region data*.

Point data: A **point** has a spatial extent characterized completely by its location; intuitively, it occupies no space and has no associated area or volume. Point data consists of a collection of *points* in a multidimensional space. Point data stored in a database can be based on direct measurements or be generated by transforming data

obtained through measurements for ease of storage and querying. **Raster data** is an example of directly measured point data and includes bit maps or pixel maps such as satellite imagery. Each pixel stores a measured value (e.g., temperature or color) for a corresponding location in space. Another example of such measured point data is medical imagery such as three-dimensional magnetic resonance imaging (MRI) brain scans. *Feature vectors* extracted from images, text, or signals such as time series are examples of point data obtained by transforming a data object. As we will see, it is often easier to use such a representation of the data, instead of the actual image or signal, to answer queries.

Region data: A **region** has a spatial extent with a location and a boundary. The location can be thought of as the position of a fixed 'anchor point' for the region, e.g., its centroid. In two dimensions, the boundary can be visualized as a line (for finite regions, a closed loop), and in three dimensions it is a surface. Region data consists of a collection of *regions*. Region data stored in a database is typically a simple geometric approximation to an actual data object. **Vector data** is the term used to describe such geometric approximations, constructed using points, line segments, polygons, spheres, cubes, etc. Many examples of region data arise in geographic applications. For instance, roads and rivers can be represented as a collection of line segments, and countries, states, and lakes can be represented as polygons. Other examples arise in computer-aided design applications. For instance, an airplane wing might be modeled as a *wire frame* using a collection of polygons (that intuitively tile the wire frame surface approximating the wing), and a tubular object may be modeled as the difference between two concentric cylinders.

Spatial queries, or queries that arise over spatial data, are of three main types: *spatial range queries*, *nearest neighbor queries*, and *spatial join queries*.

Spatial range queries: In addition to multidimensional queries such as, "Find all employees with salaries between $50,000 and $60,000 and ages between 40 and 50," we can ask queries such as, "Find all cities within 50 miles of Madison," or, "Find all rivers in Wisconsin." A spatial range query has an associated region (with a location and boundary). In the presence of region data, spatial range queries can return all regions that *overlap* the specified range or all regions that are *contained* within the specified range. Both variants of spatial range queries are useful, and algorithms for evaluating one variant are easily adapted to solve the other. Range queries occur in a wide variety of applications, including relational queries, GIS queries, and CAD/CAM queries.

Nearest neighbor queries: A typical query is, "Find the 10 cities that are nearest to Madison." We usually want the answers to be ordered by distance to Madison, i.e., by proximity. Such queries are especially important in the context of multimedia databases, where an object (e.g., images) is represented by a point, and 'similar' objects

are found by retrieving objects whose representative points are closest to the point representing the query object.

Spatial join queries: Typical examples include "Find pairs of cities within 200 miles of each other" and "Find all cities near a lake." These queries can be quite expensive to evaluate. If we consider a relation in which each tuple is a point representing a city or a lake, the above queries can be answered by a join of this relation with itself, where the join condition specifies the distance between two matching tuples. Of course, if cities and lakes are represented in more detail and have a spatial extent, both the meaning of such queries (are we looking for cities whose centroids are within 200 miles of each other or cities whose boundaries come within 200 miles of each other?) and the query evaluation strategies become more complex. Still, the essential character of a spatial join query is retained.

These kinds of queries are very common and arise in most applications of spatial data. Some applications also require specialized operations such as interpolation of measurements at a set of locations to obtain values for the measured attribute over an entire region.

26.2 APPLICATIONS INVOLVING SPATIAL DATA

There are many applications that involve spatial data. Even a traditional relation with k fields can be thought of as a collection of k-dimensional points, and as we will see in Section 26.3, certain relational queries can be executed faster by using indexing techniques designed for spatial data. In this section, however, we concentrate on applications in which spatial data plays a central role and in which efficient handling of spatial data is essential for good performance.

Geographic Information Systems (GIS) deal extensively with spatial data, including points, lines, and two- or three-dimensional regions. For example, a map contains locations of small objects (points), rivers and highways (lines), and cities and lakes (regions). A GIS system must efficiently manage two-dimensional and three-dimensional datasets. All the classes of spatial queries that we described arise naturally, and both point data and region data must be handled. Commercial GIS systems such as ArcInfo are in wide use today, and object database systems aim to support GIS applications as well.

Computer-aided design and manufacturing (CAD/CAM) systems and *medical imaging* systems store spatial objects such as surfaces of design objects (e.g., the fuselage of an aircraft). As with GIS systems, both point and region data must be stored. Range queries and spatial join queries are probably the most common queries, and **spatial integrity constraints** such as "There must be a minimum clearance of one foot

between the wheel and the fuselage" can be very useful. (CAD/CAM was a major motivation for the development of object databases.)

Multimedia databases, which contain multimedia objects such as images, text, and various kinds of time-series data (e.g., audio), also require spatial data management. In particular, finding objects similar to a given object is a common kind of query in a multimedia system, and a popular approach to answering similarity queries involves first mapping multimedia data to a collection of points called **feature vectors**. A similarity query is then converted to the problem of finding the nearest neighbors of the point that represents the query object.

In medical image databases, we have to store digitized two-dimensional and three-dimensional images such as X-rays or MRI images. Fingerprints (together with information identifying the fingerprinted individual) can be stored in an image database, and we can search for fingerprints that match a given fingerprint. Photographs from driver's licenses can be stored in a database, and we can search for faces that match a given face. Such image database applications rely on **content-based image retrieval** (e.g., find images similar to a given image). Going beyond images, we can store a database of video clips and search for clips in which a scene changes, or in which there is a particular kind of object. We can store a database of *signals* or *time-series*, and look for similar time-series. We can store a collection of text documents and search for similar documents (i.e., dealing with similar topics).

Feature vectors representing multimedia objects are typically points in a high-dimensional space. For example, we can obtain feature vectors from a text object by using a list of keywords (or concepts) and noting which keywords are present; we thus get a vector of 1s (the corresponding keyword is present) and 0s (the corresponding keyword is missing in the text object) whose length is equal to the number of keywords in our list. Lists of several hundred words are commonly used. We can obtain feature vectors from an image by looking at its color distribution (the levels of red, green, and blue for each pixel) or by using the first several coefficients of a mathematical function (e.g., the Hough transform) that closely approximates the shapes in the image. In general, given an arbitrary signal, we can represent it using a mathematical function having a standard series of terms, and approximate it by storing the coefficients of the most significant terms.

When mapping multimedia data to a collection of points, it is important to ensure that there is a measure of distance between two points that captures the notion of similarity between the corresponding multimedia objects. Thus, two images that map to two nearby points must be more similar than two images that map to two points far from each other. Once objects are mapped into a suitable coordinate space, finding similar images, similar documents, or similar time-series can be modeled as finding points that are close to each other: We map the query object to a point and look for its nearest neighbors. The most common kind of spatial data in multimedia applications

is point data, and the most common query is nearest neighbor. In contrast to GIS and CAD/CAM, the data is of high dimensionality (usually 10 or more dimensions).

26.3 INTRODUCTION TO SPATIAL INDEXES

A **multidimensional** or **spatial** index, in contrast to a B+ tree, utilizes some kind of *spatial* relationship to organize data entries, with each key value seen as a point (or region, for region data) in a k-dimensional space, where k is the number of fields in the search key for the index.

In a B+ tree index, the two-dimensional space of $\langle age, sal \rangle$ values is linearized—i.e., points in the two-dimensional domain are totally ordered—by sorting on *age* first and then on *sal*. In Figure 26.1, the dotted line indicates the linear order in which points are stored in a B+ tree. In contrast, a spatial index stores data entries based on their proximity in the underlying two-dimensional space. In Figure 26.1, the boxes indicate how points are stored in a spatial index.

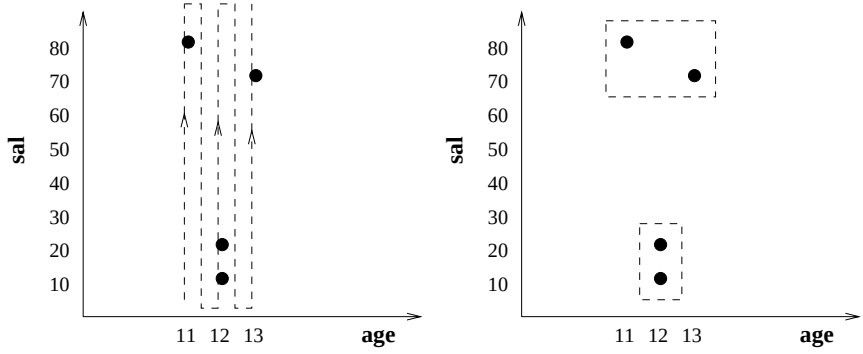

Figure 26.1 Clustering of Data Entries in B+ Tree vs. Spatial Indexes

Let us compare a B+ tree index on key $\langle age, sal \rangle$ with a spatial index on the space of *age* and *sal* values, using several example queries:

1. $age < 12$: The B+ tree index performs very well. As we will see, a spatial index will handle such a query quite well, although it cannot match a B+ tree index in this case.
2. $sal < 20$: The B+ tree index is of no use since it does not match this selection. In contrast, the spatial index will handle this query just as well as the previous selection on *age*.
3. $age < 12 \wedge sal < 20$: The B+ tree index effectively utilizes only the selection on *age*. If most tuples satisfy the *age* selection, it will perform poorly. The spatial

index will fully utilize both selections and return only tuples that satisfy both the *age* and *sal* conditions. To achieve this with B+ tree indexes, we have to create two separate indexes on *age* and *sal*, retrieve rids of tuples satisfying the *age* selection by using the index on *age* and retrieve rids of tuples satisfying the *sal* condition by using the index on *sal*, intersect these rids, then retrieve the tuples with these rids.

Spatial indexes are ideal for queries such as, "Find the 10 nearest neighbors of a given point," and, "Find all points within a certain distance of a given point." The drawback with respect to a B+ tree index is that if (almost) all data entries are to be retrieved in *age* order, a spatial index is likely to be slower than a B+ tree index in which *age* is the first field in the search key.

26.3.1 Overview of Proposed Index Structures

Many spatial index structures have been proposed. Some are primarily designed to index collections of points although they can be adapted to handle regions, and some handle region data naturally. Examples of index structures for point data include *Grid files*, *hB trees*, *KD trees*, *Point Quad trees*, and *SR trees*. Examples of index structures that handle regions as well as point data include *Region Quad trees*, *R trees*, and *SKD trees*. The above lists are far from complete; there are many variants of the above index structures and many entirely distinct index structures.

There is as yet no consensus on the 'best' spatial index structure. However, R trees have been widely implemented and found their way into commercial DBMSs. This is due to their relative simplicity, their ability to handle both point and region data, and the fact that their performance is at least comparable to more complex structures.

We will discuss three approaches that are distinct and, taken together, illustrative of many of the proposed indexing alternatives. First, we discuss index structures that rely on *space-filling curves* to organize points. We begin by discussing *Z-ordering* for point data, and then discuss Z-ordering for region data, which is essentially the idea behind Region Quad trees. Region Quad trees illustrate an indexing approach based on recursive subdivision of the multidimensional space, independent of the actual dataset. There are several variants of Region Quad trees.

Second, we discuss Grid files, which illustrate how an Extendible Hashing style directory can be used to index spatial data. Many index structures such as *Bang files*, *Buddy trees*, and *Multilevel Grid files* have been proposed, refining the basic idea. Finally, we discuss R trees, which also recursively subdivide the multidimensional space. In contrast to Region Quad trees, the decomposition of space utilized in an R tree depends upon the indexed dataset. We can think of R trees as an adaptation of the

B+ tree idea to spatial data. Many variants of R trees have been proposed, including *Cell trees*, *Hilbert R trees*, *Packed R trees*, *R* trees*, *R+ trees*, *TV trees*, and *X trees*.

26.4 INDEXING BASED ON SPACE-FILLING CURVES

Space-filling curves are based on the assumption that any attribute value can be represented with some fixed number of bits, say k bits. The maximum number of values along each dimension is therefore 2^k. We will consider a two-dimensional dataset for simplicity although the approach can handle any number of dimensions.

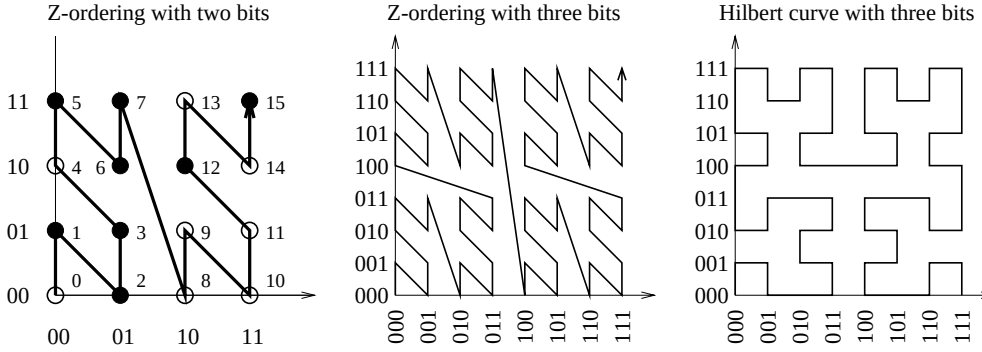

Figure 26.2 Space Filling Curves

A space-filling curve imposes a linear ordering on the domain, as illustrated in Figure 26.2. The first curve shows the **Z-ordering** curve for domains with two-bit representations of attribute values. A given dataset contains a subset of the points in the domain, and these are shown as filled circles in the figure. Domain points that are not in the given dataset are shown as unfilled circles. Consider the point with $X = 01$ and $Y = 11$ in the first curve. The point has **Z-value** 0111, obtained by interleaving the bits of the X and Y values; we take the first X bit (0), then the first Y bit (1), then the second X bit (1), and finally the second Y bit (1). In decimal representation, the Z-value 0111 is equal to 7, and the point $X = 01$ and $Y = 11$ has the Z-value 7 shown next to it in Figure 26.2. This is the eighth domain point 'visited' by the space-filling curve, which starts at point $X = 00$ and $Y = 00$ (Z-value 0).

The points in a dataset are stored in Z-value order and indexed by a traditional indexing structure such as a B+ tree. That is, the Z-value of a point is stored together with the point and is the search key for the B+ tree. (Actually, we do not have to store the X and Y values for a point if we store the Z-value, since we can compute them from the Z-value by extracting the interleaved bits.) To insert a point, we compute its Z-value and insert it into the B+ tree. Deletion and search are similarly based on computing the Z-value and then using the standard B+ tree algorithms.

The advantage of this approach over using a B+ tree index on some combination of the X and Y fields is that points are clustered together by spatial proximity in the X–Y space. Spatial queries over the X–Y space now translate into linear range queries over the ordering of Z-values and are efficiently answered using the B+ tree on Z-values.

The spatial clustering of points achieved by the Z-ordering curve is seen more clearly in the second curve in Figure 26.2, which shows the Z-ordering curve for domains with three-bit representations of attribute values. If we visualize the space of all points as four quadrants, the curve visits all points in a quadrant before moving on to another quadrant. This means that all points in a quadrant are stored together. This property holds recursively within each quadrant as well—each of the four subquadrants is completely traversed before the curve moves to another subquadrant. Thus, all points in a subquadrant are stored together.

The Z-ordering curve achieves good spatial clustering of points, but it can be improved upon. Intuitively, the curve occasionally makes long diagonal 'jumps,' and the points connected by the jumps, while far apart in the X–Y space of points, are nonetheless close in Z-ordering. The Hilbert curve, shown as the third curve in Figure 26.2, addresses this problem.

26.4.1 Region Quad Trees and Z-Ordering: Region Data

Z-ordering gives us a way to group points according to spatial proximity. What if we have region data? The key is to understand how Z-ordering recursively decomposes the data space into quadrants and subquadrants, as illustrated in Figure 26.3.

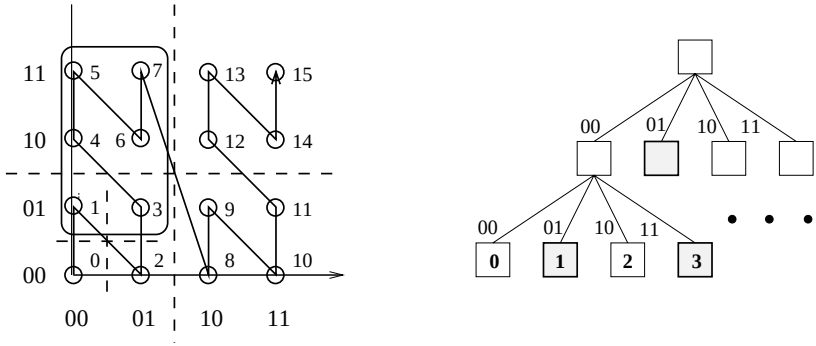

Figure 26.3 Z-Ordering and Region Quad Trees

The Region Quad tree structure corresponds directly to the recursive decomposition of the data space. Each node in the tree corresponds to a square-shaped region of the data space. As special cases, the root corresponds to the entire data space, and

some leaf nodes correspond to exactly one point. Each internal node has four children, corresponding to the four quadrants into which the space corresponding to the node is partitioned: 00 identifies the bottom left quadrant, 01 identifies the top left quadrant, 10 identifies the bottom right quadrant, and 11 identifies the top right quadrant.

In Figure 26.3, consider the children of the root. All points in the quadrant corresponding to the 00 child have Z-values that begin with 00, all points in the quadrant corresponding to the 01 child have Z-values that begin with 01, and so on. In fact, the Z-value of a point can be obtained by traversing the path from the root to the leaf node for the point and concatenating all the edge labels.

Consider the region represented by the rounded rectangle in Figure 26.3. Suppose that the rectangle object is stored in the DBMS and given the unique identifier (oid) `R`. `R` includes all points in the 01 quadrant of the root as well as the points with Z-values 1 and 3, which are in the 00 quadrant of the root. In the figure, the nodes for points 1 and 3 and the 01 quadrant of the root are shown with dark boundaries. Together, the dark nodes represent the rectangle `R`. The three records $\langle$ 0001, `R`$\rangle$, $\langle$ 0011, `R`$\rangle$, and $\langle$ 01, `R`$\rangle$ can be used to store this information. The first field of each record is a Z-value; the records are clustered and indexed on this column using a B+ tree. Thus, a B+ tree is used to implement a Region Quad tree, just as it was used to implement Z-ordering.

Note that a region object can usually be stored using fewer records if it is sufficient to represent it at a coarser level of detail. For example, rectangle `R` can be represented using two records $\langle$ 00, `R`$\rangle$ and $\langle$ 01, `R`$\rangle$. This approximates `R` by using the bottom left and top left quadrants of the root.

The Region Quad tree idea can be generalized beyond two dimensions. In k dimensions, at each node we partition the space into 2^k subregions; for $k = 2$, we partition the space into four equal parts (quadrants). We will not discuss the details.

26.4.2 Spatial Queries Using Z-Ordering

Range queries can be handled by translating the query into a collection of regions, each represented by a Z-value. (We saw how to do this in our discussion of region data and Region Quad trees.) We then search the B+ tree to find matching data items.

Nearest neighbor queries can also be handled, although they are a little trickier because distance in the Z-value space does not always correspond well to distance in the original X–Y coordinate space (recall the diagonal jumps in the Z-order curve). The basic idea is to first compute the Z-value of the query and find the data point with the closest Z-value by using the B+ tree. Then, to make sure we are not overlooking any points that are closer in the X–Y space, we compute the actual distance r between the query point and the retrieved data point and issue a range query centered at the query point

and with radius r. We check all retrieved points and return the one that is closest to the query point.

Spatial joins can be handled by extending the approach to range queries.

26.5 GRID FILES

In contrast to the Z-ordering approach, which partitions the data space independently of any one dataset, the Grid file partitions the data space in a way that reflects the data distribution in a given dataset. The method is designed to guarantee that any *point query* (a query that retrieves the information associated with the query point) can be answered in at most two disk accesses.

Grid files rely upon a **grid directory** to identify the data page containing a desired point. The grid directory is similar to the directory used in Extendible Hashing (see Chapter 10). When searching for a point, we first find the corresponding entry in the grid directory. The grid directory entry, like the directory entry in Extendible Hashing, identifies the page on which the desired point is stored, if the point is in the database. To understand the Grid file structure, we need to understand how to find the grid directory entry for a given point.

We describe the Grid file structure for two-dimensional data. The method can be generalized to any number of dimensions, but we restrict ourselves to the two-dimensional case for simplicity. The Grid file partitions space into rectangular regions using lines that are parallel to the axes. Thus, we can describe a Grid file partitioning by specifying the points at which each axis is 'cut.' If the X axis is cut into i segments and the Y axis is cut into j segments, we have a total of $i \times j$ partitions. The grid directory is an i by j array with one entry per partition. This description is maintained in an array called a **linear scale**; there is one linear scale per axis.

Figure 26.4 illustrates how we search for a point using a Grid file index. First, we use the linear scales to find the X segment to which the X value of the given point belongs and the Y segment to which the Y value belongs. This identifies the entry of the grid directory for the given point. We assume that all linear scales are stored in main memory, and therefore this step does not require any I/O. Next, we fetch the grid directory entry. Since the grid directory may be too large to fit in main memory, it is stored on disk. However, we can identify the disk page containing a given entry and fetch it in one I/O because the grid directory entries are arranged sequentially in either row-wise or column-wise order. The grid directory entry gives us the id of the data page containing the desired point, and this page can now be retrieved in one I/O. Thus, we can retrieve a point in two I/Os—one I/O for the directory entry and one for the data page.

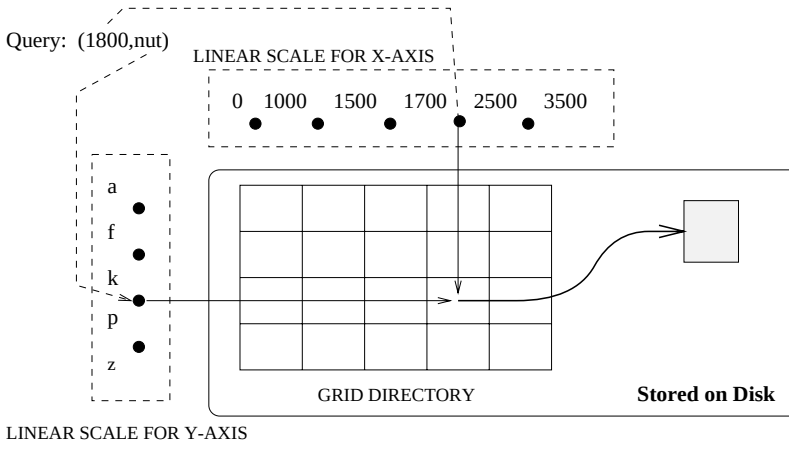

Figure 26.4 Searching for a Point in a Grid File

Range queries and nearest neighbor queries are easily answered using the Grid file. For range queries, we use the linear scales to identify the set of grid directory entries to fetch. For nearest neighbor queries, we first retrieve the grid directory entry for the given point and search the data page it points to. If this data page is empty, we use the linear scales to retrieve the data entries for grid partitions that are adjacent to the partition that contains the query point. We retrieve all the data points within these partitions and check them for nearness to the given point.

The Grid file relies upon the property that a grid directory entry points to a page that contains the desired data point (if the point is in the database). This means that we are forced to split the grid directory—and therefore a linear scale along the splitting dimension—if a data page is full and a new point is inserted to that page. In order to obtain good space utilization, we allow several grid directory entries to point to the same page. That is, several partitions of the space may be mapped to the same physical page, as long as the set of points across all these partitions fits on a single page.

Insertion of points into a Grid file is illustrated in Figure 26.5, which has four parts each illustrating a snapshot of a Grid file. Each snapshot shows just the grid directory and the data pages; the linear scales are omitted for simplicity. Initially (the top left part of the figure) there are only three points, all of which fit into a single page (A). The grid directory contains a single entry, which covers the entire data space and points to page A.

In this example, we assume that the capacity of a data page is three points. Thus, when a new point is inserted, we need an additional data page. We are also forced to split the grid directory in order to accommodate an entry for the new page. We

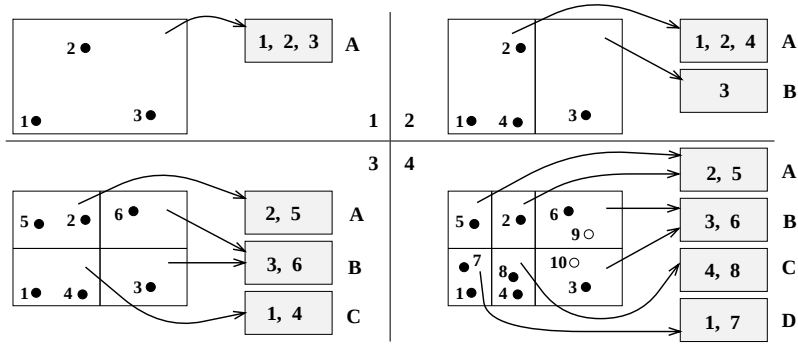

Figure 26.5 Inserting Points into a Grid File

do this by splitting along the X axis to obtain two equal regions; one of these regions points to page A and the other points to the new data page B. The data points are redistributed across pages A and B to reflect the partitioning of the grid directory. The result is shown in the top right part of Figure 26.5.

The next part (bottom left) of Figure 26.5 illustrates the Grid file after two more insertions. The insertion of point 5 forces us to split the grid directory again because point 5 is in the region that points to page A, and page A is already full. Since we split along the X axis in the previous split, we now split along the Y axis, and redistribute the points in page A across page A and a new data page, C. (Choosing the axis to split in a round-robin fashion is one of several possible splitting policies.) Observe that splitting the region that points to page A also causes a split of the region that points to page B, leading to two regions pointing to page B. Inserting point 6 next is straightforward because it is in a region that points to page B, and page B has space for the new point.

Next, consider the bottom right part of the figure. It shows the example file after the insertion of two additional points, 7 and 8. The insertion of point 7 causes page C to become full, and the subsequent insertion of point 8 causes another split. This time, we split along the X axis and redistribute the points in page C across C and the new data page D. Observe how the grid directory is partitioned the most in those parts of the data space that contain the most points—the partitioning is sensitive to data distribution, like the partitioning in Extendible Hashing, and handles skewed distributions well.

Finally, consider the potential insertion of points 9 and 10, which are shown as light circles to indicate that the result of these insertions is not reflected in the data pages. Inserting point 9 fills page B, and subsequently inserting point 10 requires a new data page. However, the grid directory does not have to be split further—points 6 and 9 can

be in page B, points 3 and 10 can go to a new page E, and the second grid directory entry that points to page B can be reset to point to page E.

Deletion of points from a Grid file is complicated. When a data page falls below some occupancy threshold, e.g., less than half-full, it must be merged with some other data page in order to maintain good space utilization. We will not go into the details beyond noting that in order to simplify deletion, there is a *convexity requirement* on the set of grid directory entries that point to a single data page: *the region defined by this set of grid directory entries must be convex.*

26.5.1 Adapting Grid Files to Handle Regions

There are two basic approaches to handling region data in a Grid file, neither of which is satisfactory. First, we can represent a region by a point in a higher dimensional space. For example, a box in two dimensions can be represented as a four-dimensional point by storing two diagonal corner points of the box. This approach does not support nearest neighbor and spatial join queries since distances in the original space are not reflected in the distances between points in the higher-dimensional space. Further, this approach increases the dimensionality of the stored data, which leads to various problems (see Section 26.7).

The second approach is to store a record representing the region object in each grid partition that overlaps the region object. This is unsatisfactory because it leads to a lot of additional records and makes insertion and deletion expensive.

In summary, the Grid file is not a good structure for storing region data.

26.6 R TREES: POINT AND REGION DATA

The R tree is an adaptation of the B+ tree to handle spatial data and it is a height-balanced data structure, like the B+ tree. The search key for an R tree is a collection of intervals, with one interval per dimension. We can think of a search key value as a *box* that is bounded by the intervals; each side of the box is parallel to an axis. We will refer to search key values in an R tree as **bounding boxes**.

A data entry consists of a pair ⟨*n-dimensional box, rid*⟩, where *rid* identifies an object and the box is the smallest box that contains the object. As a special case, the box is a point if the data object is a point instead of a region. Data entries are stored in leaf nodes. Non-leaf nodes contain index entries of the form ⟨*n-dimensional box, pointer to a child node*⟩. The box at a non-leaf node N is the smallest box that contains all boxes associated with child nodes; intuitively, it bounds the region containing all data objects stored in the subtree rooted at node N.

Figure 26.6 shows two views of an example R tree. In the first view, we see the tree structure. In the second view, we see how the data objects and bounding boxes are distributed in space.

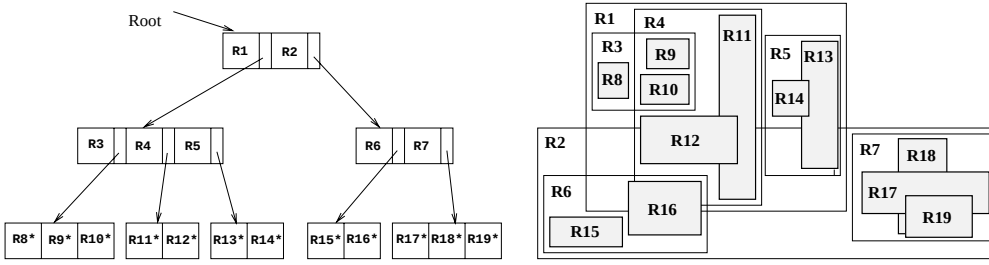

Figure 26.6 Two Views of an Example R Tree

There are 19 regions in the example tree. Regions R8 through R19 represent data objects and are shown in the tree as data entries at the leaf level. The entry R8*, for example, consists of the bounding box for region R8 and the rid of the underlying data object. Regions R1 through R7 represent bounding boxes for internal nodes in the tree. Region R1, for example, is the bounding box for the space containing the left subtree, which includes data objects R8, R9, R10, R11, R12, R13, and R14.

The bounding boxes for two children of a given node can overlap; for example, the boxes for the children of the root node, R1 and R2, overlap. This means that more than one leaf node could accommodate a given data object while satisfying all bounding box constraints. However, every data object is stored in exactly one leaf node, even if its bounding box falls within the regions corresponding to two or more higher-level nodes. For example, consider the data object represented by R9. It is contained within both R3 and R4 and could be placed in either the first or the second leaf node (going from left to right in the tree). We have chosen to insert it into the left-most leaf node; it is not inserted anywhere else in the tree. (We will discuss the criteria used to make such choices in Section 26.6.2.)

26.6.1 Queries

To search for a point, we compute its bounding box B—which is just the point—and start at the root of the tree. We test the bounding box for each child of the root to see if it overlaps the query box B, and if so we search the subtree rooted at the child. If more than one child of the root has a bounding box that overlaps B, we must search all the corresponding subtrees. This is an important difference with respect to B+ trees: *The search for even a single point can lead us down several paths in the tree.* When we get to the leaf level, we check to see if the node contains the desired point. It is possible that we do not visit *any* leaf node—this happens when the query point

is in a region that is not covered by any of the boxes associated with leaf nodes. If the search does not visit any leaf pages, we know that the query point is not in the indexed dataset.

Searches for region objects and range queries are handled similarly by computing a bounding box for the desired region and proceeding as in the search for an object. For a range query, when we get to the leaf level we must retrieve all region objects that belong there and test to see if they overlap (or are contained in, depending upon the query) the given range. The reason for this test is that even if the bounding box for an object overlaps the query region, the object itself may not!

As an example, suppose that we want to find all objects that overlap our query region, and the query region happens to be the box representing object R8. We start at the root and find that the query box overlaps R1 but not R2. Thus, we search the left subtree, but not the right subtree. We then find that the query box overlaps R3 but not R4 or R5. So we search the left-most leaf and find object R8. As another example, suppose that the query region coincides with R9 rather than R8. Again, the query box overlaps R1 but not R2 and so we search (only) the left subtree. Now we find that the query box overlaps both R3 and R4, but not R5. We therefore search the children pointed to by the entries for R3 and R4.

As a refinement to the basic search strategy, we can approximate the query region by a convex region defined by a collection of linear constraints, rather than a bounding box, and test this convex region for overlap with the bounding boxes of internal nodes as we search down the tree. The benefit is that a convex region is a tighter approximation than a box, and therefore we can sometimes detect that there is no overlap although the intersection of bounding boxes is nonempty. The cost is that the overlap test is more expensive, but this is a pure CPU cost and is negligible in comparison to the potential I/O savings.

Note that using convex regions to approximate the regions associated with nodes in the R tree would also reduce the likelihood of false overlaps—the bounding regions overlap, but the data object does not overlap the query region—but the cost of storing convex region descriptions is much higher than the cost of storing bounding box descriptions.

To search for the nearest neighbors of a given point, we proceed as in a search for the point itself. We retrieve all points in the leaves that we examine as part of this search and return the point that is closest to the query point. If we do not visit any leaves, then we replace the query point by a small box centered at the query point and repeat the search. If we still do not visit any leaves, we increase the size of the box and search again, continuing in this fashion until we visit a leaf node. We then consider all points retrieved from leaf nodes in this iteration of the search and return the point that is closest to the query point.

26.6.2 Insert and Delete Operations

To insert a data object with rid r, we compute the bounding box B for the object and insert the pair $\langle B, r\rangle$ into the tree. We start at the root node and traverse a single path from the root to a leaf (in contrast to searching, where we could traverse several such paths). At each level, we choose the child node whose bounding box needs the least enlargement (in terms of the increase in its area) to cover the box B. If several children have bounding boxes that cover B (or that require the same enlargement in order to cover B), from these children we choose the one with the smallest bounding box.

At the leaf level, we insert the object and if necessary we enlarge the bounding box of the leaf to cover box B. If we have to enlarge the bounding box for the leaf, this must be propagated to ancestors of the leaf—after the insertion is completed, the bounding box for every node must cover the bounding box for all descendants. If the leaf node does not have space for the new object, we must split the node and redistribute entries between the old leaf and the new node. We must then adjust the bounding box for the old leaf and insert the bounding box for the new leaf into the parent of the leaf. Again, these changes could propagate up the tree.

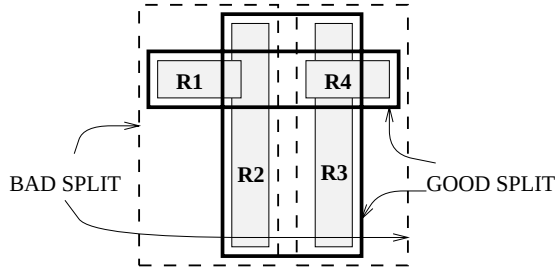

Figure 26.7 Alternative Redistributions in a Node Split

It is important to minimize the overlap between bounding boxes in the R tree because overlap causes us to search down multiple paths. The amount of overlap is greatly influenced by how entries are distributed when a node is split. Figure 26.7 illustrates two alternative redistributions during a node split. There are four regions R1, R2, R3, and R4 to be distributed across two pages. The first split (shown in broken lines) puts R1 and R2 on one page and R3 and R4 on the other page. The second split (shown in solid lines) puts R1 and R4 on one page and R2 and R3 on the other page. Clearly, the total area of the bounding boxes for the new pages is much less with the second split.

Minimizing overlap using a good insertion algorithm is very important for good search performance. A variant of the R tree called the **R* tree** introduces the concept of **forced reinserts** to reduce overlap: When a node overflows, rather than split it

immediately we remove some number of entries (about 30 percent of the node's contents works well) and reinsert them into the tree. This may result in all entries fitting inside some existing page and eliminate the need for a split. The R* tree insertion algorithms also try to minimize *box perimeters* rather than *box areas*.

To delete a data object from an R tree, we have to proceed as in the search algorithm and potentially examine several leaves. If the object is in the tree, we remove it. In principle, we can try to shrink the bounding box for the leaf containing the object and the bounding boxes for all ancestor nodes. In practice, deletion is often implemented by simply removing the object.

There is a variant called the **R+ tree** that avoids overlap by inserting an object into multiple leaves if necessary. Consider the insertion of an object with bounding box B at a node N. If box B overlaps the boxes associated with more than one child of N, the object is inserted into the subtree associated with each such child. For the purposes of insertion into child C with bounding box B_C, the object's bounding box is considered to be the overlap of B and B_C.[1] The advantage of the more complex insertion strategy is that searches can now proceed along a single path from the root to a leaf.

26.6.3 Concurrency Control

The cost of implementing concurrency control algorithms is often overlooked in discussions of spatial index structures. This is justifiable in environments where the data is rarely updated and queries are predominant. In general, however, this cost can greatly influence the choice of index structure.

We presented a simple concurrency control algorithm for B+ trees in Section 19.3.2: Searches proceed from root to a leaf obtaining shared locks on nodes; a node is unlocked as soon as a child is locked. Inserts proceed from root to a leaf obtaining exclusive locks; a node is unlocked after a child is locked if the child is not full. This algorithm can be adapted to R trees by modifying the insert algorithm to release a lock on a node only if the locked child has space *and* its region contains the region for the inserted entry (thus ensuring that the region modifications will not propagate to the node being unlocked).

We presented an index locking technique for B+ trees in Section 19.3.1, which locks a range of values and prevents new entries in this range from being inserted into the tree. This technique is used to avoid the phantom problem. Now let us consider how to adapt the index locking approach to R trees. The basic idea is to lock the index

[1] Insertion into an R+ tree involves additional details. For example, if box B is not contained in the collection of boxes associated with the children of N whose boxes B overlaps, one of the children must have its box enlarged so that B is contained in the collection of boxes associated with the children.

page that contains or would contain entries with key values in the locked range. In R trees, overlap between regions associated with the children of a node could force us to lock several (non-leaf) nodes on different paths from the root to some leaf. Additional complications arise from having to deal with changes—in particular, enlargements due to insertions—in the regions of locked nodes. Without going into further detail, it should be clear that index locking to avoid phantom insertions in R trees is both harder and less efficient than in B+ trees. Further, ideas such as forced reinsertion in R* trees and multiple insertions of an object in R+ trees make index locking prohibitively expensive.

26.6.4 Generalized Search Trees

The B+ tree and R tree index structures are similar in many respects: They are both height-balanced in which searches start at the root of the tree and proceed toward the leaves, each node covers a portion of the underlying data space, and the children of a node cover a subregion of the region associated with the node. There are important differences of course—e.g., the space is linearized in the B+ tree representation but not in the R tree—but the common features lead to striking similarities in the algorithms for insertion, deletion, search, and even concurrency control.

The **generalized search tree (GiST)** abstracts the essential features of tree index structures and provides 'template' algorithms for insertion, deletion, and searching. The idea is that an ORDBMS can support these template algorithms and thereby make it easy for an advanced database user to implement specific index structures, such as R trees or variants, without making changes to any system code. The effort involved in writing the extension methods is much less than that involved in implementing a new indexing method from scratch, and the performance of the GiST template algorithms is comparable to specialized code. (For concurrency control, more efficient approaches are applicable if we exploit the properties that distinguish B+ trees from R trees. However, B+ trees are implemented directly in most commercial DBMSs, and the GiST approach is intended to support more complex tree indexes.)

The template algorithms call upon a set of extension methods that are specific to a particular index structure and must be supplied by the implementor. For example, the search template searches all children of a node whose region is consistent with the query. In a B+ tree the region associated with a node is a range of key values, and in an R tree the region is spatial. The check to see whether a region is consistent with the query region is specific to the index structure and is an example of an extension method. As another example of an extension method, consider how to choose the child of an R tree node to insert a new entry into. This choice can be made based on which candidate child's region needs to be expanded the least; an extension method is required to calculate the required expansions for candidate children and choose the child to insert the entry into.

26.7 ISSUES IN HIGH-DIMENSIONAL INDEXING

The spatial indexing techniques that we have discussed work quite well for two- and three-dimensional datasets, which are encountered in many applications of spatial data. In some applications such as content-based image retrieval or text indexing, however, the number of dimensions can be large (tens of dimensions are not uncommon). Indexing such high-dimensional data presents unique challenges, and new techniques are required. For example, sequential scan becomes superior to R trees even when searching for a single point for datasets with more than about a dozen dimensions.

High-dimensional datasets are typically collections of points, not regions, and nearest neighbor queries are the most common kind of queries. Searching for the nearest neighbor of a query point is meaningful when the distance from the query point to its nearest neighbor is less than the distance to other points. At the very least, we want the nearest neighbor to be appreciably closer than the data point that is farthest from the query point. There is a potential problem with high-dimensional data: For a wide range of data distributions, as dimensionality d increases, the distance (from any given query point) to the nearest neighbor grows closer and closer to the distance to the farthest data point! Searching for nearest neighbors is not meaningful in such situations.

In many applications, high-dimensional data may not suffer from these problems and may be amenable to indexing. However, it is advisable to check high-dimensional datasets to make sure that nearest neighbor queries are meaningful. Let us call the ratio of the distance (from a query point) to the nearest neighbor to the distance to the farthest point the **contrast** in the dataset. We can measure the contrast of a dataset by generating a number of sample queries, measuring distances to the nearest and farthest points for each of these sample queries and computing the ratios of these distances, and taking the average of the measured ratios. In applications that call for the nearest neighbor, we should first ensure that datasets have good contrast by empirical tests of the data.

26.8 POINTS TO REVIEW

- *Spatial data* occupies a region in space called its *spatial extent*. We discussed three types of spatial queries. *Spatial range queries* specify a query region and aim to retrieve all objects that fall within or overlap the query region. *Nearest neighbor queries* specify a query point and aim to retrieve the object closest to the query point. *Spatial join queries* compute all pairs of objects that satisfy user-specified proximity constraints. **(Section 26.1)**
- There are many applications that deal with spatial data, including *Geographic Information Systems (GIS)*, which deal efficiently with two- and three-dimensional

data, and multimedia databases, which deal with high-dimensional feature vectors. A *feature vector* is a representation of the salient characteristics of an object in a high-dimensional space. Often, similarity between nonspatial objects can be expressed as the distance between feature vectors of the objects. **(Section 26.2)**

- A *multidimensional* or *spatial* index utilizes spatial relationships between data objects to organize the index. We discussed the difference between a spatial index and a B+ tree. **(Section 26.3)**
- A *space-filling curve* imposes a linear ordering on a multidimensional space. Using the linear order, objects can be indexed using a traditional index structure such as a B+ tree. If the linear ordering preserves spatial proximity, spatial queries translate into range queries on the B+ tree. We can use the recursive nature of space-filling curves to recursively partition the space; this is done in the *Region Quad tree* index structure. **(Section 26.4)**
- A *Grid file* is a spatial index structure for point data. Each dimension is partitioned into intervals that are maintained in an array called a *linear scale*. The linear scales induce a partitioning of the space into rectangles and a *grid directory* stores a pointer to the physical page associated with each rectangle. A grid file is not suitable for region data. **(Section 26.5)**
- *R trees* are height-balanced tree index structures whose search keys are *bounding boxes*. The data entries in a leaf node consist of (bounding box, rid)-pairs. A data entry in an intermediate node consists of the smallest bounding box that encloses all bounding boxes of its children. Due to overlap of bounding boxes, a query can take us down several paths in the tree. When inserting new objects, we try to minimize the overlap between bounding boxes. The *R+ tree* avoids overlap by inserting an object into multiple leaf nodes if necessary. Concurrency control in R trees is similar to concurrency control in B+ trees, although the phantom problem is more difficult to avoid. The *generalized search tree (GiST)* is a generic index template for tree-structured indexes. **(Section 26.6)**
- Indexing high-dimensional data is difficult and nearest neighbor queries are very common for such data. Nearest neighbor queries are only meaningful if the distance to the nearest neighbor differs significantly from the distance to other points—a property that often does not hold in high dimensions. The *contrast* of a data set measures its suitability for nearest neighbor queries. **(Section 26.7)**

EXERCISES

Exercise 26.1 Answer the following questions briefly.

1. How is point spatial data different from nonspatial data?
2. How is point data different from region data?

3. Describe three common kinds of spatial queries.
4. Why are nearest neighbor queries important in multimedia applications?
5. How is a B+ tree index different from a spatial index? When would you use a B+ tree index over a spatial index for point data? When would you use a spatial index over a B+ tree index for point data?
6. What is the relationship between Z-ordering and Region Quad trees?
7. Compare Z-ordering and Hilbert curves as techniques to cluster spatial data.

Exercise 26.2 Consider Figure 26.3, which illustrates Z-ordering and Region Quad trees. Answer the following questions.

1. Consider the region composed of the points with these Z-values: 4, 5, 6, and 7. Mark the nodes that represent this region in the Region Quad tree shown in Figure 26.3. (Expand the tree if necessary.)
2. Repeat the above exercise for the region composed of the points with Z-values 1 and 3.
3. Repeat it for the region composed of the points with Z-values 1 and 2.
4. Repeat it for the region composed of the points with Z-values 0 and 1.
5. Repeat it for the region composed of the points with Z-values 3 and 12.
6. Repeat it for the region composed of the points with Z-values 12 and 15.
7. Repeat it for the region composed of the points with Z-values 1, 3, 9, and 11.
8. Repeat it for the region composed of the points with Z-values 3, 6, 9, and 12.
9. Repeat it for the region composed of the points with Z-values 9, 11, 12, and 14.
10. Repeat it for the region composed of the points with Z-values 8, 9, 10, and 11.

Exercise 26.3 This exercise refers to Figure 26.3.

1. Consider the region represented by the 01 child of the root in the Region Quad tree shown in Figure 26.3. What are the Z-values of points in this region?
2. Repeat the above exercise for the region represented by the 10 child of the root and the 01 child of the 00 child of the root.
3. List the Z-values of four adjacent data points that are distributed across the four children of the root in the Region Quad tree.
4. Consider the alternative approaches of indexing a two-dimensional point dataset using a B+ tree index: (i) on the composite search key $\langle X, Y \rangle$, (ii) on the Z-ordering computed over the X and Y values. Assuming that X and Y values can be represented using two bits each, show an example dataset and query illustrating each of these cases:
   (a) The alternative of indexing on the composite query is faster.
   (b) The alternative of indexing on the Z-value is faster.

Exercise 26.4 Consider the Grid file instance with three points 1, 2, and 3 shown in the first part of Figure 26.5.

1. Show the Grid file after inserting each of these points, in the order they are listed: 6, 9, 10, 7, 8, 4, and 5.

2. Assume that deletions are handled by simply removing the deleted points, with no attempt to merge empty or underfull pages. Can you suggest a simple concurrency control scheme for Grid files?
3. Discuss the use of Grid files to handle region data.

Exercise 26.5 Answer each of the following questions independently with respect to the R tree shown in Figure 26.6. (That is, don't consider the insertions corresponding to other questions when answering a given question.)

1. Show the bounding box of a new object that can be inserted into R4 but not into R3.
2. Show the bounding box of a new object that is contained in both R1 and R6 but is inserted into R6.
3. Show the bounding box of a new object that is contained in both R1 and R6 and is inserted into R1. Which leaf node is this object placed in?
4. Show the bounding box of a new object that could be inserted into either R4 or R5, but is placed in R5 based on the principle of least expansion of the bounding box area.
5. Given an example of an object such that searching for the object takes us to both the R1 subtree and the R2 subtree.
6. Give an example query that takes us to nodes R3 and R5. (Explain if there is no such query.)
7. Give an example query that takes us to nodes R3 and R4, but not to R5. (Explain if there is no such query.)
8. Give an example query that takes us to nodes R3 and R5, but not to R4. (Explain if there is no such query.)

BIBLIOGRAPHIC NOTES

Several multidimensional indexing techniques have been proposed. These include Bang files [245], Grid files [494], hB trees [426], KDB trees [548], Pyramid trees [67] Quad trees[565], R trees [295], R∗ trees [59], R+ trees, the TV tree, and the VA file [673]. [273] discusses how to search R trees for regions defined by linear constraints. Several variations of these, and several other distinct techniques, have also been proposed; Samet's text [566] deals with many of them. A good recent survey is [252].

The use of Hilbert curves for linearizing multidimensional data is proposed in [225]. [100] is an early paper discussing spatial joins. [317] proposes a generalized tree index that can be specialized to obtain many of the specific tree indexes mentioned earlier. Concurrency control and recovery issues for this generalized index are discussed in [386]. [318] discusses the complexity of indexing schemes, in particular range queries, and [80] discusses the problems arising with high-dimensionality. [220] provides a good overview of how to search multimedia databases by content. A recent trend is towards spatiotemporal applications, such as tracking moving objects [686].