23. Metric data structures

Learning objectives:

- organizing the embedding space versus organizing its contents
- quadtrees and octtrees. grid file. two-disk-access principle
- simple geometric objects and their parameter spaces
- region queries of arbitrary shape
- approximation of complex objects by enclosing them in simple containers

Organizing the embedding space versus organizing its contents

Most of the data structures discussed so far organize the set of elements to be stored depending primarily, or even exclusively, on the relative values of these elements to each other and perhaps on their order of insertion into the data structure. Often, the only assumption made about these elements is that they are drawn from an ordered domain, and thus these structures support only *comparative search* techniques: the search argument is compared against stored elements. The shape of data structures based on comparative search varies dynamically with the set of elements currently stored; it does not depend on the static domain from which these elements are samples. These techniques organize the particular contents to be stored rather than the embedding space.

The data structures discussed in this chapter mirror and organize the domain from which the elements are drawn—much of their structure is determined before the first element is ever inserted. This is typically done on the basis of fixed points of reference which are independent of the current contents, as inch marks on a measuring scale are independent of what is being measured. For this reason we call data structures that organize the embedding space *metric data structures*. They are of increasing importance, in particular for *spatial data*, such as needed in computer-aided design or geographic data processing. Typically, these domains exhibit a much richer structure than a mere order: In two- or three-dimensional Euclidean space, for example, not only is *order* defined along *any line* (not just the coordinate axes), but also *distance* between any two points. Most queries about spatial data involve the absolute position of elements in space, not just their relative position among each other. A typical query in graphics, for example, asks for the first object intercepted by a given ray of light. Computing the answer involves absolute position (the location of the ray) and relative order (nearest along the ray). A data structure that supports direct access to objects according to their position in space can clearly be more efficient than one based merely on the relative position of elements.

The terms "organizing the embedding space" and "organizing its contents" suggest two extremes along a spectrum of possibilities. As we have seen in previous chapters, however, many data structures are hybrids that combine features from distinct types. This is particularly true of metric data structures: They always have aspects of address computation needed to locate elements in space, and they often use list processing techniques for efficient memory utilization.

Radix trees, tries

We have encountered binary radix trees, and a possible implementation, in chapter 22 in the section “Extendible hashing”. Radix trees with a branching factor, or *fan-out*, greater than 2 are ubiquitous. The Dewey decimal classification used in libraries is a radix tree with a fan-out of 10. The hierarchical structure of many textbooks, including this one, can be seen as a radix tree with a fan-out determined by how many subsections at depth d + 1 are packed into a section at depth d.

As another example, consider *tries*, a type of radix tree that permits the retrieval of variable-length data. As we traverse the tree, we check whether or not the node we are visiting has any successors. Thus the trie can be very long along certain paths. As an example, consider a trie containing words in the English language. In Exhibit 23.1 below, the four words 'a', 'at', 'ate', and 'be' are shown explicitly. The letter 'a' is a word and is the first letter of other words. The field corresponding to 'a' contains the value 1, signaling that we have spelled a valid word, and there is a pointer to longer words beginning with 'a'. The letter 'b' is not a word, thus is marked by a 0, but it is the beginning of many words, all found by following its pointer. The string 'aa' is neither a word nor the beginning of a word, so its field contains 0 and its pointer is 'nil'.

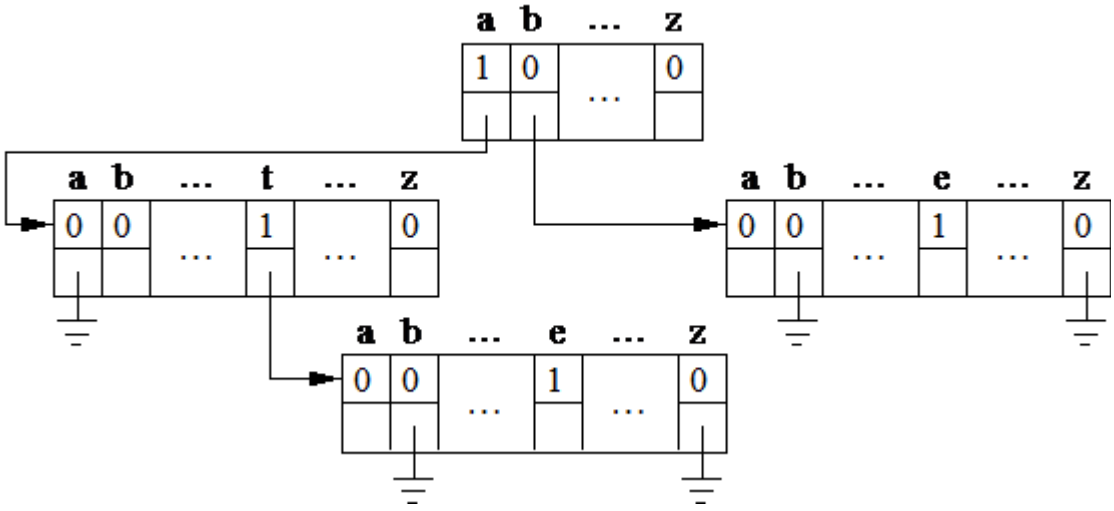

Exhibit 23.1: A radix tree over the alphabet of letters stores (prefixes of) words.

Only a few words begin with 'ate', but among these there are some long ones, such as 'atelectasis'. It would be wasteful to introduce eight additional nodes, one for each of the characters in 'lectasis', just to record this word, without making significant use of the fan-out of 26 provided at each node. Thus tries typically use an "overflow technique" to handle long entries: The pointer field of the prefix 'ate' might point to a text field that contains '(ate-)lectasis' and '(ate-)lier'.

Quadtrees and octtrees

Consider a square recursively partitioned into quadrants. Exhibit 23.2 23.2 shows such a square partitioned to the depth of 4. There are 4 quadrants at depth 1, separated by the thickest lines; $4 \cdot 4$ (sub-)quadrants separated by slightly thinner lines; 4^3 (sub-sub-)quadrants separated by yet thinner lines; and finally, $4^4 = 256$ leaf quadrants separated by the thinnest lines. The partitioning structure described is a *quadtree*, a particular type of radix tree of fan-out 4. The root corresponds to the entire square, its 4 children to the 4 quadrants at depth 1, and so on, as shown in the Exhibit 23.2.

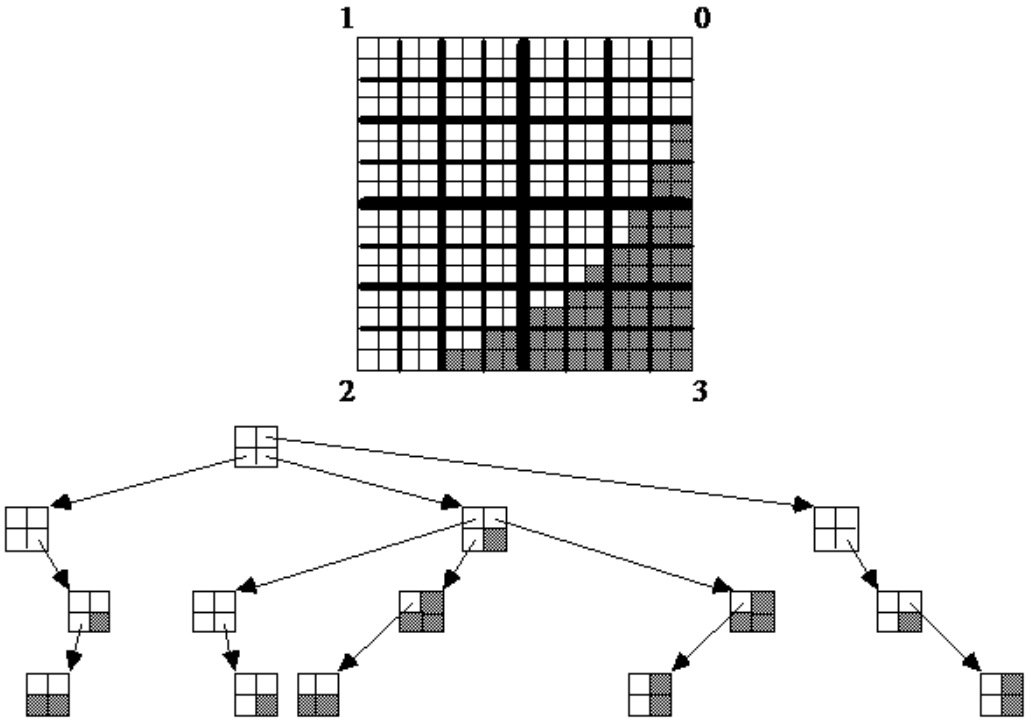

Exhibit 23.2: A quarter circle digitized on a 16 · 16 grid, and its representation as a 4-level quadtree.

A quadtree is the obvious two-dimensional analog of the one-dimensional binary radix tree we have seen. Accordingly, quadtrees are frequently used to represent, store, and process spatial data, such as images. The figure shows a quarter circle, digitized on a 16 · 16 grid of pixels. This image is most easily represented by a 16 · 16 array of bits. The quadtree provides an alternative representation that is advantageous for images digitized to a high level of resolution. Most graphic images in practice are digitized on rectangular grids of anywhere from hundreds to thousands of pixels on a side: for example, 512 · 512. In a quadtree, only the largest quadrants of constant color (black or white, in our example) are represented explicitly; their subquadrants are implicit.

The quadtree in Exhibit 23.2 is interpreted as follows. Of the four children of the root, the northwest quadrant, labeled 1, is simple: entirely white. This fact is recorded in the root. The other three children, labeled 0, 2, and 3, contain both black and white pixels. As their description is not simple, it is contained in three quadtrees, one for each quadrant. Pointers to these subquadtrees emanate from the corresponding fields of the root.

The southwestern quadrant labeled 2 in turn has four quadrants at depth 2. Three of these, labeled 2.0, 2.1, and 2.2, are entirely white; no pointers emanate from the corresponding fields in this node. Subquadrant 2.3 contains both black and white pixels; thus the corresponding field contains a pointer to a sub-subquadtree.

In this discussion we have introduced a notation to identify every quadrant at any depth of the quadtree. The root is identified by the null string; a quadrant at depth d is uniquely identified by a string of d radix-4 digits. This string can be interpreted in various ways as a number expressed in base 4. Thus accessing and processing a quadtree is readily reduced to arithmetic.

Breadth-first addressing

Label the root 0, its children 1, 2, 3, 4, its grand children 5 through 20, and so on, one generation after the other.

```
0
  1          2              3             4
     5 6 7 8   9 10 11 12    13 14 15 16   17 18 19 20
```

Notice that the children of any node i are $4 \cdot i + 1$, $4 \cdot i + 2$, $4 \cdot i + 3$, $4 \cdot i + 4$. The parent of node i is $(i - 1)$ div 4. This is similar to the address computation used in the heap of "Implicit data structures", a binary tree where each node i has children $2 \cdot i$ and $2 \cdot i + 1$; and the parent of node i is obtained as i div 2.

Exercise

The string of radix 4 digits along a path from the root to any node is called the *path address* of this node. Interpret the path address as an integer, most significant digit first. These integers label the nodes at depth $d > 0$ consecutively from 0 to $4^d - 1$. Devise a formula that transforms the path address into the breadth-first address. This formula can be used to store a quadtree as a one-dimensional array.

Data compression

The representation of an image as a quadtree is sometimes much more compact than its representation as a bit map. Two conditions must hold for this to be true:

1. The image must be fairly large, typically hundreds of pixels on a side.
2. The image must have large areas of constant value (color).

The quadtree for the quarter circle above, for example, has only 14 nodes. A bit map of the same image requires 256 bits. Which representation requires more storage? Certainly the quadtree. If we store it as a list, each node must be able to hold four pointers, say 4 or 8 bytes. If a pointer has value 'nil', indicating that its quadrant needs no refinement, we need a bit to indicate the color of this quadrant (white or black), or a total of 4 bits. If we store the quadtree breadth-first, no pointers are needed as the node relationships are expressed by address computation; thus a node is reduced to four three-valued fields ('white', 'black', or 'refine'), conveniently stored in 8 bits, or 1 byte. This implicit data structure will leave many unused holes in memory. Thus quadtrees do not achieve data compression for small images.

Octtrees

Exactly the same idea for three-dimensional space as quadtrees are for two-dimensional space: A cube is recursively partitioned into eight octants, using three orthogonal planes.

Spatial data structures: objectives and constraints

Metric data structures are used primarily for storing spatial data, such as points and simple geometric objects embedded in a multidimensional space. The most important objectives a spatial data structure must meet include:

1. Efficient handling of large, dynamically varying data sets in interactive applications
2. Fast access to objects identified in a fully specified query
3. Efficient processing of proximity queries and region queries of arbitrary shape
4. A uniformly high memory utilization

Achieving these objectives is subject to many constraints, and results in trade-offs.

Managing disks. By "large data set" we mean one that must be stored on disk; only a small fraction of the data can be kept in central memory at any one time. Many data structures can be used in central memory, but the choice is much more restricted when it comes to managing disks because of the well-known "memory speed gap"

phenomenon. Central memory is organized in small physical units (a byte, a word) with access times of approximately 1 microsecond, 10^{-6} second. Disks are organizein large physical blocks (512 bytes to 5kilobytes) with access times ranging from 10 to 100 milliseconds (10^{-2} to 10^{-1} second). Compared to central memory, a disk delivers data blocks typically 10^3 times larger with a delay 10^4 times greater. In terms of the data rate delivered to the central processing unit:

$$\frac{\text{size of data block read}}{\text{access time}}$$

the disk is a storage device whose effectiveness is within an order of magnitude of that of central memory. The large size of a physical disk block is a potential source of inefficiency that can easily reduce the useful data rate of a disk a hundredfold or a thousandfold. Accessing a couple of bytes on disk, say a pointer needed to traverse a list, takes about as long as accessing the entire disk block. Thus the game of managing disks is about *minimizing the number of disk access*es.

Dynamically varying data. The majority of computer applications today are interactive. That means that insertions, deletions, and modifications of data are at least as frequent as operations that merely process fixed data. Data structures that entail a systematic degradation of performance with continued use (such as ever-lengthening overflow chains, or an ever-increasing number of cells marked "deleted" in a conventional hash table) are unsuitable. Only structures that automatically adapt their shape to accommodate ever-changing contents can provide uniform response times.

Instantaneous response. Interactive use of computers sets another major challenge for data management: the goal of providing "instantaneous response" to a fully specified query. "Fully" specified means that every attribute relevant for the search has been provided, and that at most one element satisfies the query. Imagine the user clicking an icon on the screen, and the object represented by the icon appears instantaneously. In human terms, "instantaneous" is a well-defined physiological quantity, namely, about of a second, the limit of human time resolution. Ideally, an interactive system retrieves any single element fully specified in a query within 0.1 second.

Two-disk-access principle. We have already stated that in today's technology, a disk access typically takes from tens of milliseconds. Thus the goal of retrieving any single element in 0.1 second translates into "retrieve any element in at most a few disk accesses". Fortunately, it turns out that useful data structure can be designed that access data in a two-step process: (1) access the correct portion of a directory, and (2) access the correct data bucket. Under the assumption that both data and directory are so large that they are stored on disk, we call this the *two-disk-access principle*.

Proximity queries and region queries of arbitrary shape. The simplest example of a proximity query is the operation 'next', which we have often encountered in one-dimensional data structure traversals: Given a pointer to an element, get the next element (the successor or the predecessor) according to the order defined on the domain. Another simple example is an interval or range query such as "get all x between 13 and 17". This generalizes directly to k-dimensional *orthogonal range queries* such as the two-dimensional query "get all (x_1, x_2) with $13 \le x_1 < 17$ and $3 \le x_2 < 4$". In geometric computation, for example, many other instances of proximity queries are important, such as the "nearest neighbor" (in any direction), or intersection queriesamong objects. Region queries of arbitrary shape (not just rectangular) are able to express a variety of geometric conditions.

Uniformly high memory utilization. Any data structure that adapts its shape to dynamically changing contents is likely to leave "unused holes" in storage space: space that is currently unused, and that cannot conveniently be used for other purposes because it is fragmented. We have encountered this phenomenon in multiway trees such as B-trees and in hash tables. It is practically unavoidable that dynamic data structures use their allocated space to less than 100%, and an average space utilization of 50% is often tolerable. The danger to avoid is a built-in bias that drives space utilization toward 0 when the file shrinks—elements get deleted but their space is not relinquished. The grid file, to be discussed next, achieves an average memory utilization of about 70% regardless of the mix of insertions or deletions.

The grid file

The grid file is a metric data structure designed to store points and simple geometric objects in multidimensional space so as to achieve the objectives stated above. This section describes its architecture, access and update algorithms, and properties. More details can be found in [NHS 84] and [Hin 85].

Scales, directory, buckets

Consider as an example a two-dimensional domain: the Cartesian product $X_1 \times X_2$, where $X_1 = 0 .. 1999$ is a subrange of the integers, and $X_2 = a .. z$ is the ordered set of the 26 characters of the English alphabet. Pairs of the form (x_1, x_2), such as (1988, w), are elements from this domain.

The *bit map* is a natural data structure for storing a set S of elements from $X_1 \times X_2$. It may be declared as

```
var  T: array[X1, X2] of boolean;
```

with the convention that

$$T[x_1, x_2] = \text{true} \quad \Leftrightarrow \quad (x_1, x_2) \in S.$$

Basic set operations are performed by direct access to the array element corresponding to an element: find(x_1, x_2) is simply the boolean expression $T[x_1, x_2]$; insert(x_1, x_2) is equivalent to $T[x_1, x_2]$:= 'true', delete(x_1, x_2) is equivalent to $T[x_1, x_2]$:= 'false'. The bit map for our small domain requires an affordable 52k bits. Bit maps for realistic examples are rarely affordable, as the following reasoning shows. First, consider that x and y are just keys of records that hold additional data. If space is reserved in the array for this additional data, an array element is not a bit but as many bytes as are needed, and all the absent records, for elements $(x_1, x_2) \notin S$, waste a lot of storage. Second, most domains are much larger than the example above: the three-dimensional Euclidean space, for example, with elements (x, y, z) taken as triples of 32-bit integers, or 64-bit floating-point numbers, requires bit maps of about 10^{30} and 10^{60} bits, respectively. For comparison's sake: a large disk has about 10^{10} bits.

Since large bit maps are extremely sparsely populated, they are amenable to data compression. The grid file is best understood as a practical data compression technique that stores huge, sparsely populated bit maps so as to support direct access. Returning to our example, imagine a historical database indexed by the year of birth and the first letter of the name of scientists: thus we find 'John von Neumann' under (1903, v). Our database is pictured as a cloud of points in the domain shown in Exhibit 23.3; because we have more scientists (or at least, more records) in recent years, the density increases toward the right. Storing this database implies packing the records into buckets of fixed capacity to hold c (e.g. c = 3) records. The figure shows the domain partitioned by orthogonal hyperplanes into box-shaped grid cells, none of which contains more than c points.

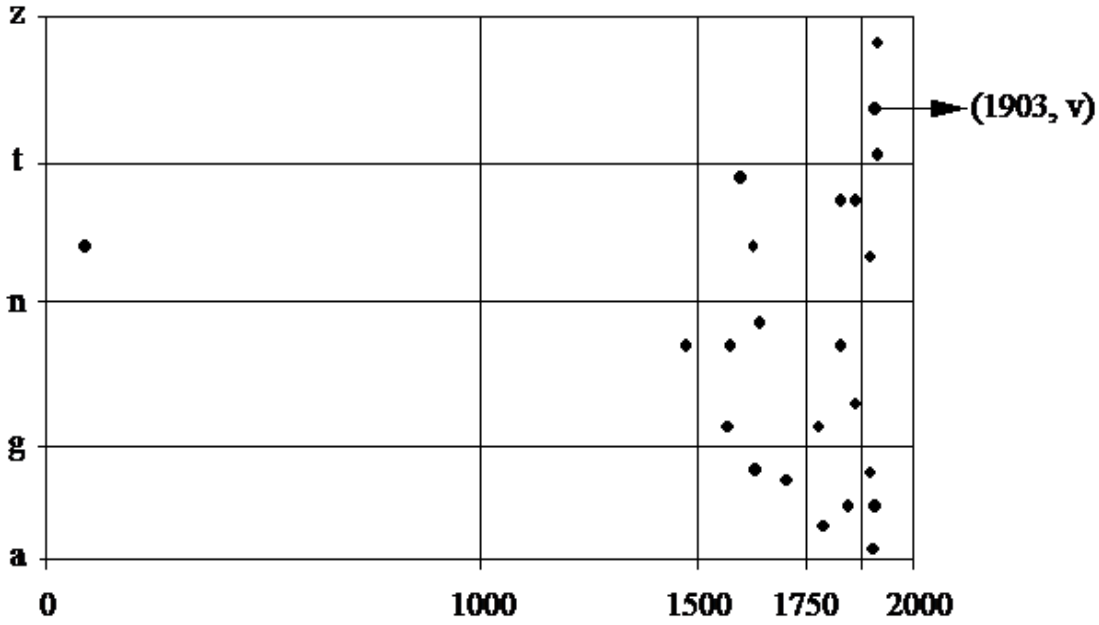

Exhibit 23.3: Cells of a grid partition adapt their size so that no cell is populated by more than c points.

A grid file for this database contains the following components:

- *Linear scales* show how the domain is currently partitioned.
- The *directory* is an array whose elements are in one-to-one correspondence with the grid cells; each entry points to a *data bucket* that holds all the records of the corresponding grid cell.

Access to the record (1903, v) proceeds through three steps:

1. Scales transform key values to array indices: (1903, v) becomes (5, 4). Scales contain small amounts of data, which is kept in central memory; thus this step requires no disk access.
2. The index tuple (5, 4) provides direct access to the correct element of the directory. The directory may be large and occupy many pages on disk, but we can compute the address of the correct directory page and in one disk access retrieve the correct directory element.
3. The directory element contains a pointer (disk address) of the correct data bucket for (1903, v), and the second disk access retrieves the correct record: [(1903, v), John von Neumann ...].

Disk utilization

The grid file does not allocate a separate bucket to each grid cell—that would lead to an unacceptably low disk utilization. Exhibit 23.4 suggests, for example, that the two grid cells at the top right of the directory share the same bucket. How this bucket sharing comes about, and how it is maintained through splitting of overflowing buckets, and merging sparsely populated buckets, is shown in the following.

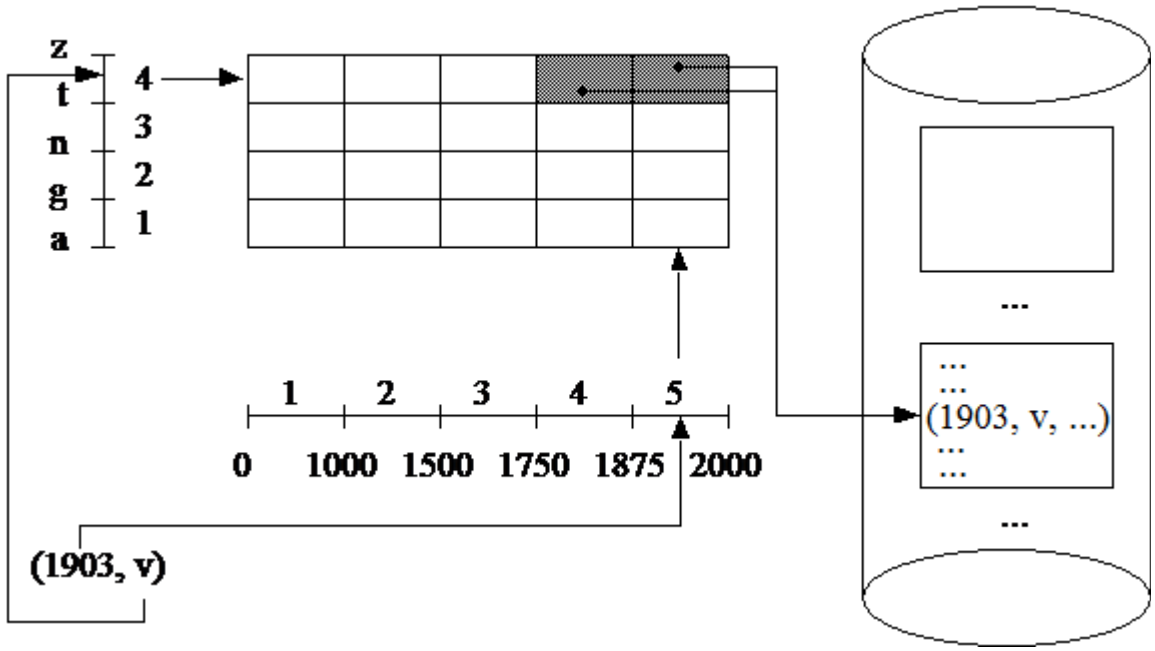

Exhibit 23.4: The search for a record with key values (1903, v) starts with the scales and proceeds via the directory to the correct data bucket on disk.

The dynamics of splitting and merging

The dynamic behavior of the grid file is best explained by tracing an example: we show the effect of repeated insertions in a two-dimensional file. Instead of showing the grid directory, whose elements are in one-to-one correspondence with the grid blocks, we draw the bucket pointers as originating directly from the grid blocks.

Initially, a single bucket A, of capacity c = 3 in our example, is assigned to the entire domain (Exhibit 23.5). When bucket A overflows, the domain is split, a new bucket B is made available, and those records that lie in one half of the space are moved from the old bucket to the new one (Exhibit 23.6). If bucket A overflows again, its grid block (i.e. the left half of the space) is split according to some splitting policy: We assume the simplest splitting policy of alternating directions. Those records of A that lie in the lower-left grid block of Exhibit 23.7 are moved to a new bucket C. Notice that as bucket B did not overflow, it is left alone: Its region now consists of two grid blocks. For effective memory utilization it is essential that in the process of refining the grid partition we need not necessarily split a bucket when its region is split.

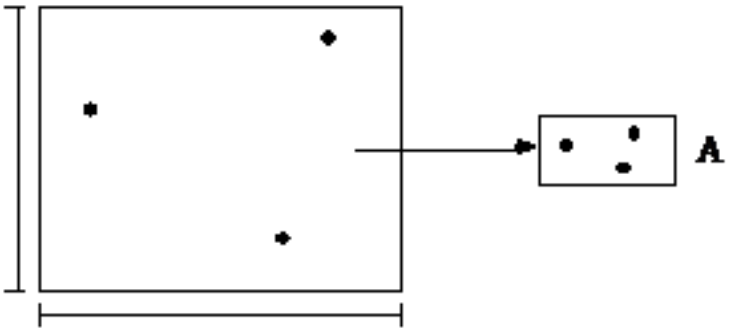

Exhibit 23.5: A growing grid file starts with a single bucket allocated to the entire key space.

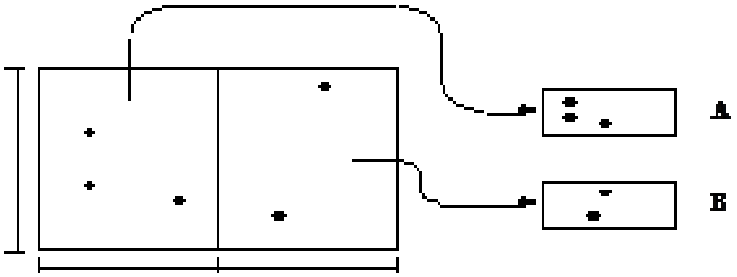

Exhibit 23.6: An overflowing bucket triggers a refinement of the space partition.

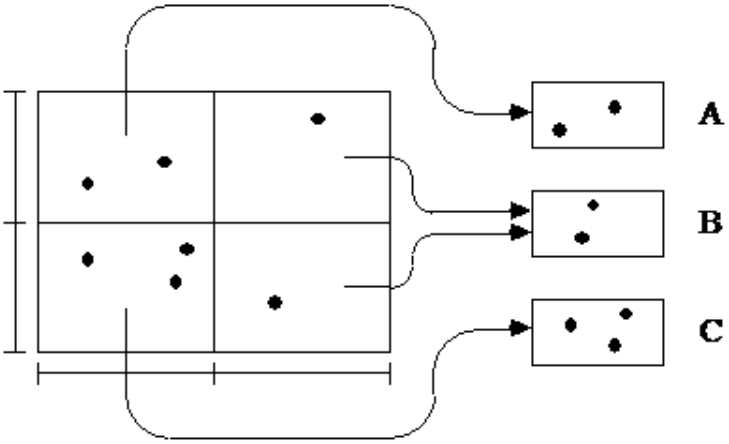

Exhibit 23.7: Bucket A has been split into A and C, but the contents of B remain unchanged.

Assuming that records keep arriving in the lower-left corner of the space, bucket C will overflow. This will trigger a further refinement of the grid partition as shown in Exhibit 23.8, and a splitting of bucket C into C and D. The history of repeated splitting can be represented in the form of a binary tree, which imposes on the set of buckets currently in use (and hence on the set of regions of these buckets) a *twin system* (also called a *buddy system*): Each bucket and its region have a unique twin from which it split off. In Exhibit 23.8, C and D are twins, the pair (C, D) is A's twin, and the pair (A, (C, D)) is B's twin.

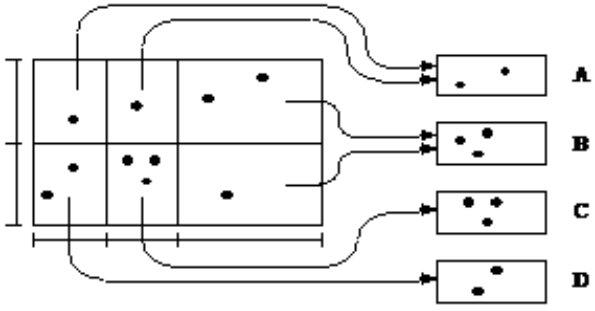

Exhibit 23.8: Bucket regions that span several cells ensure high disk utilization.

Deletions trigger merging operations. In contrast to one-dimensional storage, where it is sufficient to merge buckets that split earlier, merging policies for multidimensional grid files need to be more general in order to maintain a high occupancy.

Simple geometric objects and their parameter spaces

Consider a class of simple spatial objects, such as aligned rectangles in the plane (i.e. with sides parallel to the axes). Within its class, each object is defined by a small number of parameters. For example, an aligned rectangle is determined by its center (cx, cy) and the half-length of each side, dx and dy.

An object defined within its class by k parameters can be considered to be a point in a k-dimensional parameter space. For example, an aligned rectangle becomes a point in four-dimensional space. All of the geometric and topological properties of an object can be deduced from the class it belongs to and from the coordinates of its corresponding point in parameter space.

Different choices of the parameter space for the same class of objects are appropriate, depending on characteristics of the data to be processed. Some considerations that may determine the choice of parameters are:

1. *Distinction between location parameters and extension parameters.* For some classes of simple objects it is reasonable to distinguish location parameters, such as the center (cx, cy) of an aligned rectangle, from extension parameters, such as the half-sides dx and dy. This distinction is always possible for objects that can be described as Cartesian products of spheres of various dimensions. For example, a rectangle is the product of two one-dimensional spheres, a cylinder the product of a one-dimensional and a two-dimensional sphere. Whenever this distinction can be made, cone-shaped search regions generated by proximity queries as described in the next section have a simple intuitive interpretation: The subspace of the location parameters acts as a "mirror" that reflects a query.
2. *Independence of parameters, uniform distribution.* As an example, consider the class of all intervals on a straight line. If intervals are represented by their left and right endpoints, lx and rx, the constraint $lx \leq rx$ restricts all representations of these intervals by points (lx, rx) to the triangle above the diagonal. Any data structure that organizes the embedding space of the data points, as opposed to the particular set of points that must be stored, will pay some overhead for representing the unpopulated half of the embedding space. A coordinate transformation that distributes data all over the embedding space leads to more efficient storage. The phenomenon of nonuniform data distribution can be worse than this. In most applications, the building blocks from which complex objects are built are much smaller than the space in which they are embedded, as the size of a brick is small compared to the size of a house. If so, parameters such as lx and rx that locate boundaries of an object are highly dependent on each other. Exhibit 23.9 shows short intervals on a long line clustering along the diagonal, leaving large regions of a large embedding space unpopulated; whereas the same set of intervals represented by a location parameter cx and an extension parameter dx fills a smaller embedding space in a much more uniform way. With the assumption of bounded dx, this data distribution is easier to handle.

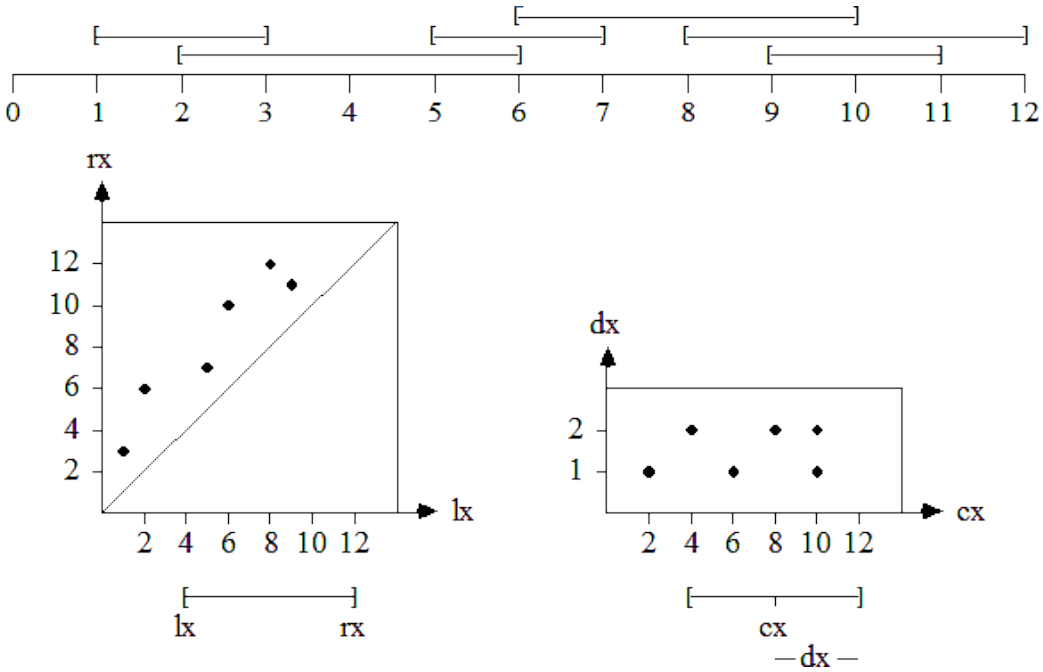

Exhibit 23.9: A set of intervals represented in two different parameter spaces.

Region queries of arbitrary shape

Intersection is a basic component of other proximity queries, and thus deserves special attention. CAD design rules, for example, often require different objects to be separated by some minimal distance. This is equivalent to requiring that objects surrounded by a rim do not intersect. Given a subset Γ of a class of simple spatial objects with parameter space H, we consider two types of queries:

- point query Given a query point q, find all objects $A \in \Gamma$ for which $q \in A$.
- point set query Given a query set Q of points, find all objects $A \in \Gamma$ that intersect Q.

Point query. For a query point q compute the region in H that contains all points representing objects in Γ that overlap q.

1. Consider the class of intervals on a straight line. An interval given by its center cx and its half length dx overlaps a point q with coordinate qx if and only if $cx - dx \leq qx \leq cx + dx$.
2. The class of aligned rectangles in the plane (with parameters cx, cy, dx, dy) can be treated as the Cartesian product of two classes of intervals, one along the x-axis, the other along the y-axis (Exhibit 23.10). All rectangles that contain a given point q are represented by points in four-dimensional space that lie in the Cartesian product of two point-in-interval query regions. The region is shown by its projections onto the cx-dx plane and the cy-dy plane.

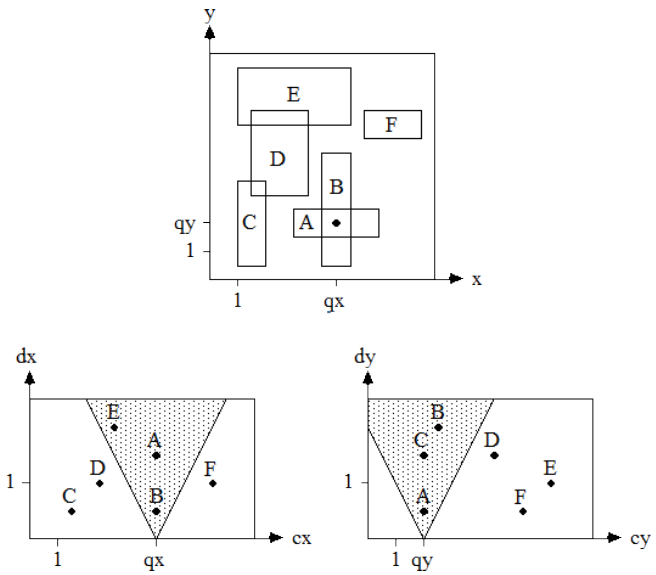

Exhibit 23.10: A set of aligned rectangles represented as a set of points in a four-dimensional parameter space. A point query is transformed into a cone-shaped region query.

3. Consider the class of circles in the plane. We represent a circle as a point in three-dimensional space by the coordinates of its center (cx, cy) and its radius r as parameters. All circles that overlap a point q are represented in the corresponding three-dimensional space by points that lie in the cone with vertex q shown in Exhibit 23.11. The axis of the cone is parallel to the r-axis (the extension parameter), and its vertex q is considered a point in the cx-cy plane (the subspace of the location parameters).

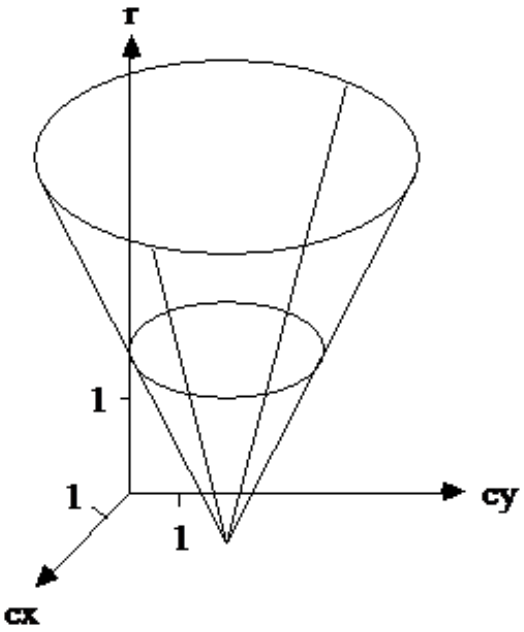

Exhibit 23.11: Search cone for a point query for circles in the plane.

Point set query. Given a query set Q of points, the region in H that contains all points representing objects A ∈ Γ that intersect Q is the *union* of the regions in H that results from the point queries for each point q ∈ Q. The union of cones is a particularly simple region in H if the query set Q is a simple spatial object.

1. Consider the class of intervals on a straight line. An interval i = (cx, dx) intersects a query interval Q = (cq, dq) if and only if its representing point lies in the shaded region shown in Exhibit 23.12; this region is given by the inequalities cx – dx ≤ cq + dq and cx + dx ≥ cq – dq.

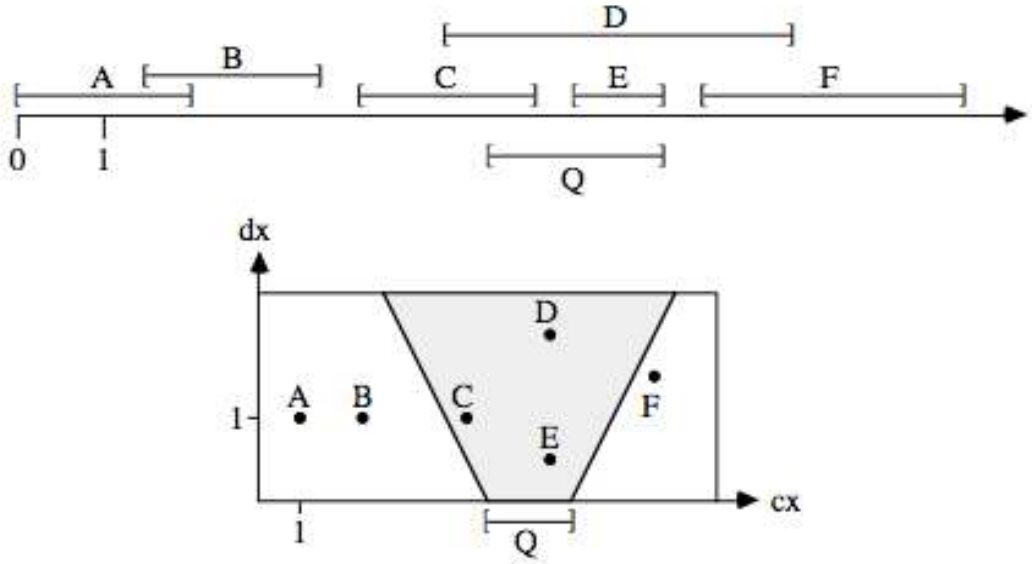

Exhibit 23.12: An interval query, as a union of point queries, again gets transformed into a search cone.

2. The class of aligned rectangles in the plane is again treated as the Cartesian product of two classes of intervals, one along the x-axis, the other along the y-axis. If Q is also an aligned rectangle, all rectangles that intersect Q are represented by points in four-dimensional space lying in the Cartesian product of two interval intersection query regions.
3. Consider the class of circles in the plane. All circles that intersect a line segment L are represented by points lying in the cone-shaped solid shown in Exhibit 23.13. This solid is obtained by embedding L in the cx-cy plane, the subspace of the location parameters, and moving the cone with vertex at q along L.

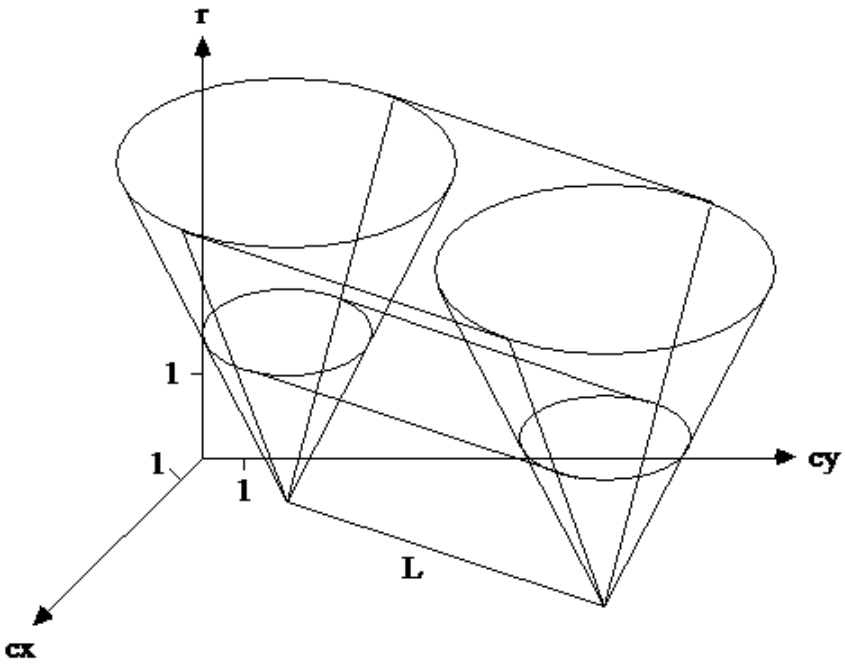

Exhibit 23.13: Search region as a union of cones.

Evaluating region queries with a grid file

We have seen that proximity queries on spatial objects lead to search regions significantly more complex than orthogonal range queries. The grid file allows the evaluation of irregularly shaped search regions in such a way that the complexity of the region affects CPU time but not disk accesses. The latter limits the performance of a data base implementation. A query region Q is matched against the scales and converted into a set I of index tuples that refer to entries in the directory. Only after this preprocessing do we access disk to retrieve the correct pages of the directory and the correct data buckets whose regions intersect Q (Exhibit 23.14).

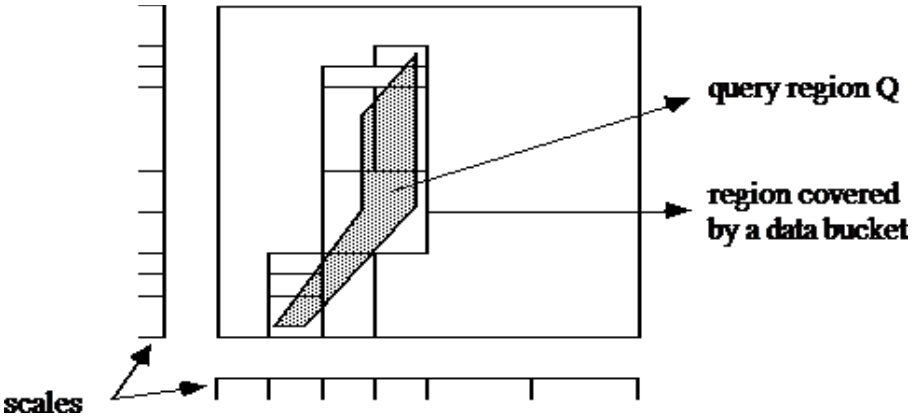

Exhibit 23.14: The cells of a grid partition that overlap an arbitrary query region Q are determined by merely looking up the scales.

Interaction between query processing and data access

The point of the two preceding sections was to show that in a metric data structure, intricate computations triggered by proximity queries can be preprocessed to a remarkable extent *before* the objects involved are retrieved.

Query preprocessing may involve a significant amount of computation based on small amounts of auxiliary data—the scales and the query—that are kept in central memory. The final access of data from disk is highly selective—data retrieved has a high chance of being part of the answer.

Contrast this to an approach where an object can be accessed only by its name (e.g. the part number) because the geometric information about its location in space is only included in the record for this object but is not part of the accessing mechanism. In such a database, all objects might have to be retrieved in order to determine which ones answer the query. Given that disk access is the bottleneck in most database applications, it pays to preprocess queries as much as possible in order to save disk accesses.

The integration of query processing and accessing mechanism developed in the preceding sections was made possible by the assumption of simple objects, where each instance is described by a small number of parameters. What can we do when faced with a large number of irregularly shaped objects?

Complex, irregularly shaped spatial objects can be represented or approximated by simpler ones in a variety of ways, for example: *decomposition*, as in a quad tree tessellation of a figure into disjoint raster squares; representation as a *cover* of overlapping simple shapes; and enclosing each object in a *container* chosen from a class of simple shapes. The container technique allows efficient processing of proximity queries because it preserves the most important properties for proximity-based access to spatial objects, in particular: It does not break up the object into components that must be processed separately, and it eliminates many potential tests as unnecessary (if two containers don't intersect, the objects within won't either). As an example, consider finding all polygons that intersect a given query polygon, given that each of them is enclosed in a simple container such as a circle or an aligned rectangle. Testing two polygons for intersection is an expensive operation compared to testing their containers for intersection. The cheap container test excludes most of the polygons from an expensive, detailed intersection check.

Any approximation technique limits the primitive shapes that must be stored to one or a few types: for example, aligned rectangles or boxes. An instance of such a type is determined by a few parameters, such as coordinates of its center and its extension, and can be considered to be a point in a (higher-dimensional) parameter space. This transformation reduces object storage to point storage, increasing the dimensionality of the problem without loss of information. Combined with an efficient multi-dimensional data structure for *point* storage it is the basis for an effective implementation of databases for spatial objects.

Exercises

1. Draw three quadtrees, one for each of the $4 \cdot 8$ pixel rectangles A, B and C outlined in Exhibit 23.15.

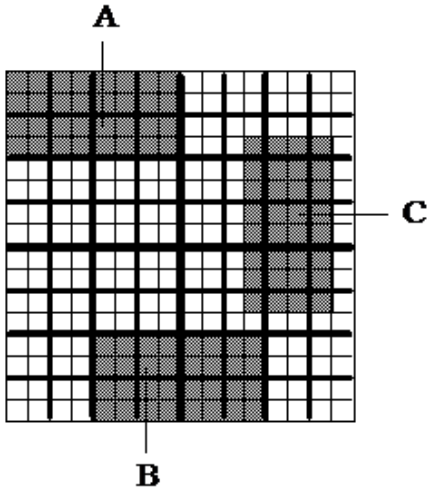

Exhibit 23.15: The location of congruent objects greatly affects the complexity of a quadtree representation.

2. Consider a grid file that stores points lying in a two-dimensional domain: the Cartesian product X1 × X2, where X_1 = 0 .. 15 and X_2 = 0 .. 15 are subranges of the integers. Buckets have a capacity of two points.

   (a) Insert the points (2, 3), (13, 14), (3, 5), (6, 9), (10, 13), (11, 5), (14, 9), (7, 3), (15, 11), (9, 9), and (11, 10) into the initially empty grid file and show the state of the scales, the directory, and the buckets after each insert operation. Buckets are split such that their shapes remain as quadratic as possible.

   (b) Delete the points (10, 13), (9, 9), (11, 10), and (14, 9) from the grid file obtained in a) and show the state of the scales, the directory, and the buckets after each delete operation. Assume that after deleting a point in a bucket this bucket may be merged with a neighbor bucket if their joint occupancy does not exceed two points. Further, a boundary should be removed from its scale if there is no longer a bucket that is split with respect to this boundary.

   (c) Without imposing further restrictions a deadlock situation may occur after a sequence of delete operations: No bucket can merge with any of its neighbors, since the resulting bucket region would no longer be rectangular. In the example shown in Exhibit 23.16 the shaded ovals represent bucket regions. Devise a merging policy that prevents such deadlocks from occurring in a two-dimensional grid file.

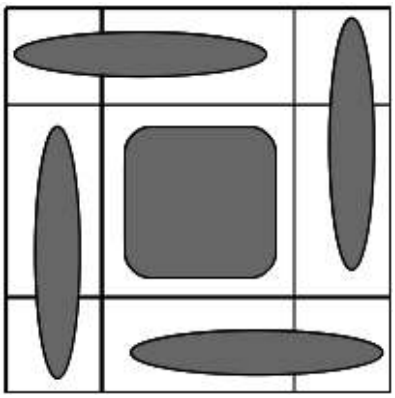

Exhibit 23.16: This example shows bucket regions that cannot be merged pairwise.

3. Consider the class of circles in the plane represented as points in three-dimensional parameter space as proposed in chapter 23 in the section “Region queries of arbitrary shape”. Describe the search regions in the parameter space (a) for all the circles intersecting a given circle C, (b) for all the circles contained in C, and (c) for all the circles enclosing .