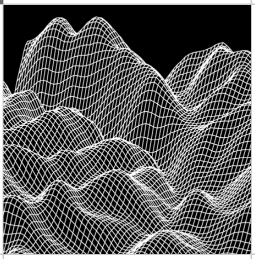
6 Spatial Access Methods



"The wolf took a short cut out of the woods, and soon came to the cottage of Red Riding Hood's grandmother."

Red Riding Hood
Illustrated by R. Andre (New York, McLoughlin Bros., 1888)

C O N T E N T S

6.1 Issues in SAM Design 204
6.2 Space-Driven Structures 208
6.3 Data-Driven Structures: The R-Tree 237
6.4 Bibliographic Notes 261

In order to efficiently process spatial queries, one needs specific access methods relying on a data structure called an *index*. These access methods accelerate the access to data; that is, they reduce the set of objects to be looked at when processing a query. Numerous examples of queries selecting or joining objects were discussed in Chapter 3. If the selection or join criteria are based on descriptive attributes, a classical index such as the B-tree can be used. However, we shall see that different access methods are needed for spatial queries, in which objects are selected according to their location in space.

Point and *window queries* are examples of spatial queries (see Figure 1.7). In such queries, we look for objects whose geometry contains a point or overlaps a rectangle. *Spatial join* is another important example. Given two sets of objects, we want to keep pairs of objects satisfying some spatial relationships. Intersection, adjacency, containment, and North-West are examples of spatial predicates a pair of objects to be joined must satisfy.

As seen in Chapter 5, processing a spatial query leads to the execution of complex and costly geometric operations. For common operations such as point queries, sequentially scanning and checking whether each object of a large collection contains a point involves a large number of disk accesses and the repeated expensive evaluation of geometric predicates. Hence, both the time-consuming geometric algorithms and the large volume of spatial collections stored on secondary storage motivate the design of efficient spatial access methods (SAMs), which reduce the set of objects to be processed. With a SAM, one expects a time that is logarithmic in the collection size, or even smaller. In most cases, the SAM uses an explicit structure, called a *spatial index*. The collection of objects for which an index is built is said to be *indexed*.

We restrict our attention to objects whose spatial component is defined in the 2D plane. We will show some SAMs for points, but we focus on the indexing of lines and polygons. In this case, instead of indexing the object geometries themselves—whose shape might be complex—we usually index a simple approximation of the geometry. The most commonly used approximation is the minimum bounding rectangle of the objects' geometry, or *minimal bounding box*, denoted *mbb* in the following. By using the *mbb* as the geometric key for constructing spatial indices, we save the cost of evaluating expensive geometric predicates during index traversal (a geometric test against an *mbb* can be processed in constant time). A second important motivation is to use constant-size entries, a feature that simplifies the design of spatial structures.

A spatial index is built on a collection of *entries*; that is, on pairs [*mbb, oid*] where *oid* identifies the object whose minimal bounding box is *mbb*. We assume that *oid* allows us to access directly the page that contains the physical representation of the object; that is, the values of the descriptive attributes and the value of the spatial component.

An operation involving a spatial predicate on a collection of objects indexed on their *mbb* is performed in two steps. The first step, called *filter step*, selects the objects whose *mbb* satisfies the spatial predicate. This step consists of traversing the index, applying the spatial test on the *mbb*. The output is a set of *oid*s.

An *mbb* might satisfy a query predicate, whereas the exact geometry does not. Then the objects that pass the filter step are a superset of the solution. In a second step, called *refinement step*, this superset is sequentially scanned, and the spatial test is done on the actual geometries of objects whose *mbb* satisfied the filter step. This geometric operation is costly, but is executed only on a limited number of objects. The objects that pass the filter step but not the refinement step are called *false drops*. In this chapter, we will not look at the refinement step, which is the same whatever the SAM is. The geometric operations in the refinement step have been studied in Chapter 5. We concentrate here on the design of efficient SAMs for the filter step.

In off-the-shelf database management systems, the most common access methods use B-trees and hash tables, which guarantees that the number of input/output operations (I/Os) to access data is respectively logarithmic and constant in the collection size, for exact search queries. A typical example of exact search is "Find County (whose name=) San Francisco." Unfortunately, these access methods cannot be used in the context of spatial data. As an example, take the B-tree. This structure indexes a collection on a *key;* that is, an attribute value of every object of the collection. The B-tree relies on a total order on the key domain, usually the order on natural numbers, or the lexicographic order on strings of characters. Because of this order, interval queries are efficiently answered.

A convenient order for geometric objects in the plane is one that preserves object proximity. Two objects *close* in the plane should be close in the index structure. Take, for example, objects inside a rectangle. They are close in the plane. It is desirable that their representation in the index also be close, so that a window query can be efficiently answered. Unfortunately, there is no such total order. The same limitation holds for hashing. Therefore, a large number of SAMs were designed to try as much as possible to preserve object proximity. The proposals are based on some heuristics, such as rectangle inclusion.

We present in this chapter two families of indices representative of two major trends; namely, the *grid* and the *R-tree*. These structures are simple, robust, and efficient, at least on well-behaved data. The bibliographic notes at the end of the chapter mention a variety of other structures. For each family, we successively study the following operations:

- *Index construction:* Insertion of one entry or of a collection of entries.
- *Search operations:* Point and window queries. The *search space* is assumed to be a rectangular subset of the 2D plane with sides parallel to the x and y axes.

The design of efficient SAMs is crucial for optimizing not only point and window queries but spatial operations such as join. We postpone the discussion of spatial join to Chapter 7.

The chapter is organized as follows. Section 6.1 discusses the main issues involved in the design of multidimensional access methods. We provide the assumptions on which the design of access methods is based, specify what we require from SAMs, and describe a simple classification. Grid-based structures are presented in Section 6.2, and the R-tree in Section 6.3. A variety of other structures and some more advanced material is discussed in the bibliographic notes of Section 6.4.

6.1 ISSUES IN SAM DESIGN

The design of SAMs relies on the same fundamental assumptions that originated the development of B-trees or hashing techniques for accelerating data access in classical databases. First, the collection size is much larger than the space available in main memory. Second, the access time to disks is long compared to that of random access main memory. Finally, objects are grouped in *pages*, which constitute the unit of transfer between memory and disk. Typical page sizes range from 1K to 4K bytes. Reading/writing one page from/to disk is referred to as an input/output operation, or I/O for short.

Once a page has been loaded into main memory, it is available for a further query unless the memory utilized for this page has been in the meantime reused for reading another page from disk. Then when accessing a particular object, two cases may happen: either the page containing the object is already in main memory or there is a *page fault* and the page has to be loaded from disk.

In traditional database indexing, central processing unit (CPU) time is assumed to be negligible compared to I/O time. The efficiency of the SAM is therefore measured in number of I/Os. In spatial database indexing, the CPU time should be taken into account in some situations. However, we still keep the assumption that it is negligible compared to I/O time. In addition, we do not take into account the caching of pages in main memory and assume the worst-case conditions. This means that each time a page is accessed, it is *not* present in main memory and a page fault occurs.

6.1.1 *What Is Expected of a SAM?*

Given these assumptions, a SAM should fulfill the following requirements:

- *Time complexity.* It should support exact (point) and range (window) search in sublinear time.
- *Space complexity.* Its size should be comparable to that of the indexed collection.

The first requirement states that accessing with the SAM a small subset of the collection must take less time than a sequential scan whose complexity is linear in the collection size. The space complexity criterion serves to measure how well the structure maps onto the physical resources of the underlying system. Indeed, if the indexed collection occupies n pages, the index size should be of the order of n.

Another important aspect of a SAM is *dynamicity.* It must accept insertions of new objects and deletions of existing ones, and adapt to any growth or shrink of the indexed collection (or to a nonuniform distribution of objects) without performance loss. It is a difficult task to get a structure robust enough to support any statistical distribution of rectangle shape, size, or location. For most of the structures studied in the following, on average, the index traversal is efficient. That is, its access time (measured in number of I/Os) is logarithmic in the collection size. However, there always exists some worst case in which the index is not performing.

The requirement that the structure adapts smoothly to dynamic insertions is important but less essential. One can design efficient structures that only support a *static* construction on a collection of objects known in advance.

6.1.2 *Illustration with a B+Tree*

The following is a concrete illustration of these properties using the classical variant of a B-tree, called a *B+tree*. Figure 6.1a depicts a B+tree index built on the set of keys {2, 3, 7, 9, 10, 13, 15}. It is a balanced tree in which each node is a disk page and stores *entries*, which are pairs of $[key, ptr]$ values, where ptr points either to an object whose key attribute has for a value key (leaf nodes) or to a child node (internal nodes). Here, we assume that each page stores at most four entries. All entries in a node are ordered; for each entry e, subtrees on the left index objects o such that $o.key \leq e.key$, and subtrees on the right index objects such that $o.key > e.key$.

Given a key value val, an exact search is performed by traversing the tree top-down, starting from the root and, at each node, choosing the subtree left to the smallest entry e such that $val \leq e.key$. If val is greater than the largest entry in the node, then the rightmost subtree is chosen. In such a tree, all leaves are at the same depth (it is balanced).

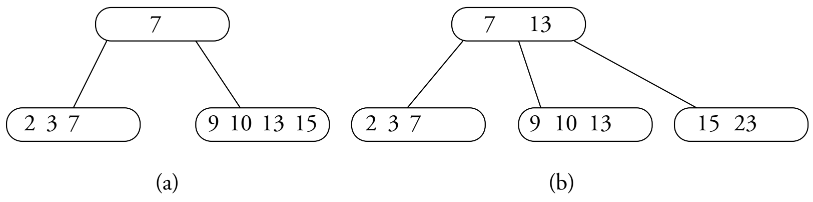


Figure 6.1 B-trees: A B+tree before (a) and after insertion of $[23, oid]$ (b).

Then, the number of pages that must be read is equal to the depth of the tree, which is logarithmic in the size of the indexed data set.

Now consider the insertion of an entry $e = [23, oid]$. The insertion occurs in the second leaf (see Figure 6.1b). This leaf is full and must be split. Half the entries (the leftmost part) are kept in the page, the remaining half and e are inserted in a new leaf (in a new page), and a new entry with key 13 is inserted in the parent node. The result is shown in Figure 6.1b. The tree is still balanced and, in spite of the split, each node is still half full.

In summary, the B+tree is a nice illustration of the properties to be expected from a secondary memory structure. First (*time complexity*), the number of I/Os to access an object is logarithmic in the collection size. It is the length of a path from the root to a tree leaf. Second (*space complexity*), at least half the space needed to store the index is actually used. This ensures that the size of the index is comparable to the size of the indexed collection (it is smaller in most cases, because the key used to construct the B-tree is much smaller than an entire record in the data set). Finally (*dynamic updates*), the structure preserves its properties through random insertions and deletions, and adapts itself to any distribution.

6.1.3 Space-Driven Versus Data-Driven SAMs

All properties of the B-tree that permit efficient indexing rely on the existence of a total order on the key values. There is no simple adaptation of the technique to spatial data. We must use a different construction principle, which is not based on the ordering of objects on their coordinates. A large number of SAMs exist that follow one of two approaches:

- *Space-driven structures:* These are based on partitioning of the embedding 2D space into rectangular cells, independently of the distribution of the objects (points or *mbb*) in the two-dimensional plane. Objects are mapped to the cells according to some geometric criterion.
- *Data-driven structures:* These structures are organized by partitioning the set of objects, as opposed to the embedding space. The partitioning adapts to the objects' distribution in the embedding space.

We choose to present well-known families of SAMs for each category. Some of them have recently been implemented in commercial systems. The first SAM to be presented includes the *grid file* and *linear structures*. These are both space-driven methods.

The second set of SAMs presented in this chapter belongs to the data-driven family of R-trees. Similar to the B-tree, the R-tree is balanced, provides restructuring operations that allow one to reorganize the structure locally, and adapts itself to nonuniform distributions. We present the original R-tree structure and two variants, called R*tree and R+tree.

The lack of robustness of SAMs with respect to the statistical properties of the objects in the two-dimensional plane is true for all SAMs presented here, whether they are space driven or data driven. When the rectangle distribution is not too skewed, their behavior is satisfactory and the response time is close to optimal. Unfortunately, one can always find a skewed distribution for which the performance drastically degrades. In other words, although the B-tree is a worst-case optimal structure, this is not true anymore for the common multidimensional access methods.

Finally, recall that in the case of one-dimensional and two-dimensional objects, SAMs only accelerate the access to the objects's *mbb;* that is, they allow a fast filter step, which must be followed by a refinement step in which the exact geometry of the object (line or polygon) is checked against the query predicate. This issue is addressed in Chapter 7.

6.2 SPACE-DRIVEN STRUCTURES

Two families of SAMs are presented in this section: the *grid file* and *linear structures*. The *grid file* is an improvement of the *fixed grid* initially designed for indexing points. We will successively present in Section 6.2.1 both structures for points and their adaptation to rectangle indexing. The grid file was initially designed for indexing objects on the value of several attributes. Unlike the B-tree, it is a *multikey index* that supports queries on any combination of these attributes. *Linear structures* enable a simple integration with the B+tree of existing database management systems. Such a technique has been used in spatial extensions of commercial relational DBMSs (e.g., Oracle, Chapter 8). Two linear structures will be successively presented: the linear quadtree (Section 6.2.2) and the z-ordering tree (Section 6.2.3).

6.2.1 The Grid File

We start with the description of the grid file structure for point indexing, and then look in detail at its adaptation to indexing *mbb* (rectangles). We first present a simple variant, called *fixed grid*.

THE FIXED GRID

The search space is decomposed into rectangular *cells*. The resulting regular grid is an $n_x \times n_y$ array of equal-size cells. Each cell c is associated with a disk page. Point P is assigned to cell c if the rectangle *c.rect* associated with cell c contains P. The objects mapped to a cell c are sequentially stored in the page associated with c.

Figure 6.2 depicts a fixed grid indexing a collection of 18 points. The search space has for an origin the point with coordinates ($x0$, $y0$). The index requires a 2D array $DIR[1 : n_x, 1 : n_y]$ as a *directory*. Each element $DIR[i, j]$ of the directory contains the address $PageID$ of the page storing the points assigned to cell $c_{i,j}$. If $[S_x, S_y]$ is the 2D size of the search space, each cell's rectangle has size $[S_x/n_x, S_y/n_y]$. We sketch in the following the algorithms for point insertion and window query (the algorithm for point deletion is straightforward):

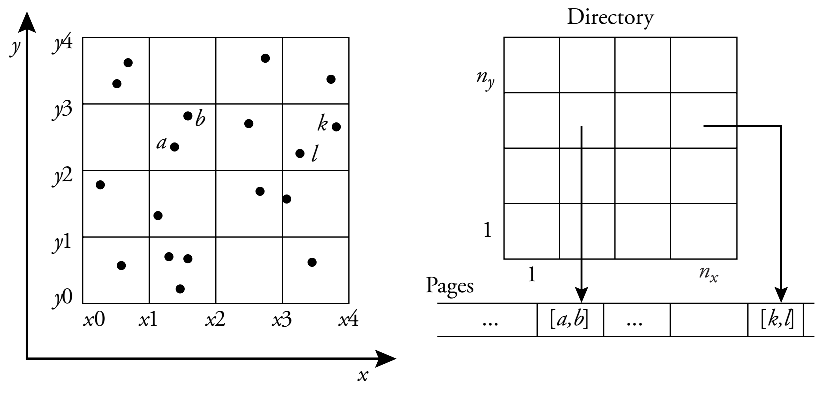


Figure 6.2 Fixed grid.

- *Inserting* $P(a, b)$: compute $i = (a - x_0)/(S_x/n_x) + 1$ and $j = (b - y_0)/(S_y/n_y) + 1$, and then read page $DIR[i, j].PageID$ and insert P.
- *Point query:* given the point argument $P(a, b)$, get the page as for insertion, read the page, scan the entries, and check whether one is P.
- *Window query:* compute the set S of cells c such that *c.rect* overlaps the query argument window W;[23] read, for each cell $c_{i,j}$ in S, page $DIR[i, j].PageID$ and return the points in the page that are contained in the argument window W.

Point query is efficient in this context. Assuming the directory resides in central memory, a point query requires a single I/O. The number of I/Os for a window query depends on the number of cells the window intersects; that is, it is proportional to the area of the window.

The grid resolution depends on the number N of points to be indexed. Given a page capacity of M points, one can create a fixed grid with at least N/M cells. Each cell contains, on the average, less than M objects. A cell to which more than M points are assigned overflows. *Overflow pages* are then chained to the initial page. For instance, assuming that each page can store at most four entries, points satisfying a skewed distribution will be indexed as illustrated in Figure 6.3. k, l, m, n, and o are assigned to a cell that overflows. k, l, m, and n are stored

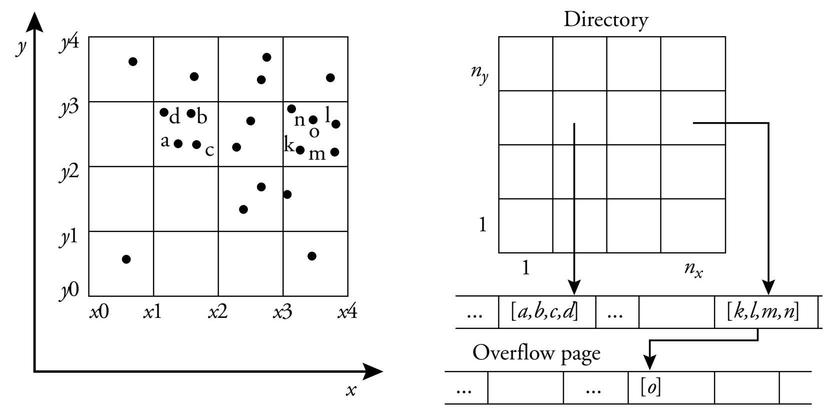


Figure 6.3 Page overflow in the fixed grid.

23. Algorithm GF-WindowQuery (see p. 218) explains how this set can be computed.

in page p. o is stored in an overflow page linked to p. Then a point query might take up to q I/Os if the point is stored in the qth page of an overflow chain.

If the points are uniformly distributed in the search space, the phenomenon previously described seldom occurs and the overflow chains are not long. This is obviously not the case with skewed point distributions. In the worst case, all points fall into the same cell and the structure is a linear list of pages. Searching a point then takes linear time, and the structure fails to meet the basic requirement of sublinear time complexity. Overflows also occur when the data set size varies significantly. Even if the point distribution stays uniform, repeated insertions of new points result in a large number of overflow pages. Conversely, deletions might result in almost empty pages. In summary, under certain types of distribution, the fixed grid fails to satisfy the requirements expected of an index in secondary memory.

POINT INDEXING WITH THE GRID FILE

In the grid file, as in the fixed grid, one page is associated with each cell, but when a cell overflows, it is split into two cells and the points are assigned to the new cell they fall into. Then cells are of different size and the partition adapts to the point distribution. Three data structures are necessary (see Figure 6.4):

- The *directory DIR* is a 2D array that references pages associated with cells. The structure is similar to that of the fixed grid, the important difference being that two adjacent cells can reference the same page. In Figure 6.4, the directory is not explicitly represented. Pointers to disk pages are overlaid on the corresponding cells of the grid.
- Two *scales* S_x and S_y are linear arrays describing the partition of a coordinate axis into intervals. Each value in one of the scales (say, S_x) represents a boundary in the partition of the search space along the related dimension (here, the x axis).

The construction of the structure is best described using an example. Figure 6.4 illustrates four steps in the construction of a grid file on a collection of points. For the sake of clarity, we assume that M, the capacity of a page, is 4.

In step A, the structure contains four objects only. The directory is a single cell associated with page $p1$. In step B, points a, b, and c are

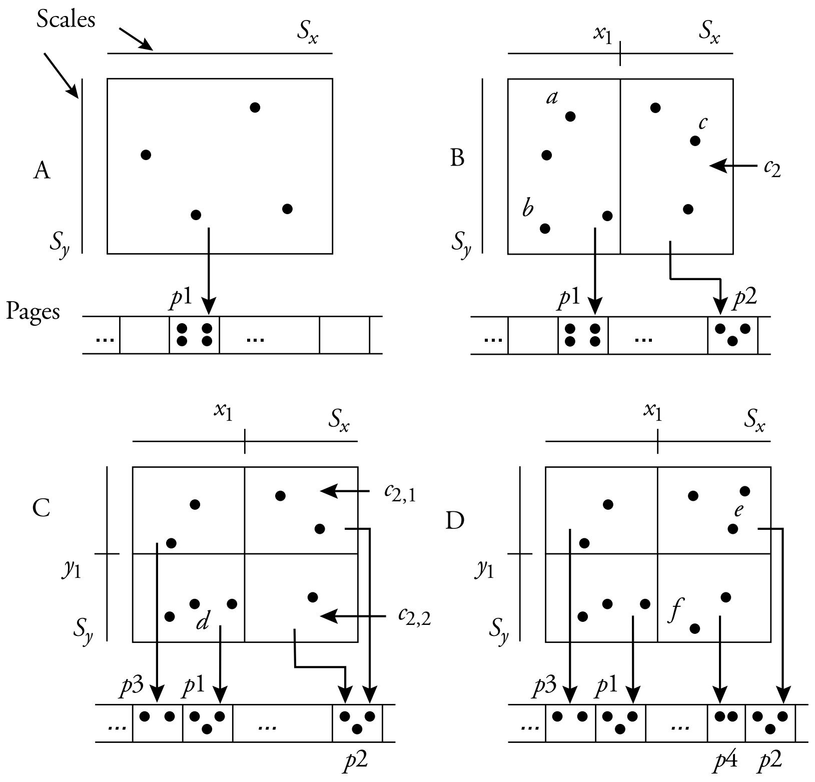


Figure 6.4 Insertions into a grid file.

successively inserted. When inserting a, $p1$ overflows: a vertical split is performed, a new data page $p2$ is allocated, and the five points are distributed among $p1$ and $p2$. The vertical split is reported in scale S_x as value x_1. a and b are inserted in $p1$, and c in $p2$.

In step C, point d is inserted. It should be stored in page $p1$, but $p1$ is full. A horizontal split is performed, reported on scale S_y as a value y_1. The five points are then distributed among pages $p1$ and $p3$.

Observe that the split also implies splitting all cells in the same S_y interval as y_1, although those cells do not overflow. However, the nonoverflowing split cells keep referencing the same pages. Cell c_2 is replaced by two new cells, $c_{2,1}$ and $c_{2,2}$, which both point to $p2$ (Figure 6.4c).

Finally, in step D, e and f are inserted. Page $p2$ overflows and a new page ($p4$) must be allocated. No interval split is necessary in this case, because a division was already defined in the *DIR* structure at value y_1. The points of cell $c_{2,1}$ are stored in $p2$, whereas the points of cell $c_{2,2}$ are stored in the new page $p4$. In summary, when inserting point P, three cases must be considered:

- *No cell splits.* This is the simplest case: point P falls into a page that is not full. Then the page is read and the point is inserted into it in sequence.
- *Cell split and no directory splits.* The point P to be inserted falls into cell c, and page p associated with c is full *but* referenced by several (at least two) distinct cells. In that case, a new page p' is allocated and assigned to c, and objects in p contained in *c.rect*, as well as the point to be inserted, are moved to p'. The directory entry corresponding to c is updated with p' (see step D).
- *Cell split and directory split.* This is the most complex case. Page p referenced by cell $c_{i,j}$ into which P falls is full, and there are no other cells referencing p. Then $c_{i,j}$*.rect* is split along either x or y.

  Assume, without loss of generality, that the split is done along the x axis. Then the cell projection on S_x is split into two intervals and a new abscissa is inserted in S_x, with rank $i + 1$.[24] The old $c_{i,j}$ cell generated two half-size cells $c_{i,j}$ and $c_{i+1,j}$. A new page p' is created and assigned to $c_{i+1,j}$. It remains to distribute the points of the full page between page p (points that fall into $c_{i,j}$*.rect*) and page p' (points falling into $c_{i+1,j}$*.rect*).

  All previous columns in *DIR* with rank l, $l > i$ are switched to $l + 1$, and a new column with index $i + 1$ is created. Now, for all $k \neq j$, the new cell $c_{i+1,k}$ is assigned the same page as $c_{i,k}$ (see step C).

The grid file overcomes major limits of the fixed grid. A point search can always be performed with two disk accesses (one for accessing the page p referenced in the directory and one for the data page associated with one entry in p). In addition, the structure is dynamic, and it presents a reasonable behavior with respect to space utilization (although degenerate cases may occur). However, a shortcoming of the method is that for large data sets the number of cells in the direc-

24. The simplest choice is to split the x interval into two equal parts.

tory might be so large that the directory does not fit anymore in main memory.

rectangle indexing with grids

We now turn to the more complex case of rectangle (*mbb*) indexing. The simplest way of mapping rectangles to cells is to assign the rectangles to the cells that overlap the rectangles. Three cases may happen. The rectangle either contains the cell, intersects it, or is contained in it. In the two former cases, the rectangle is assigned to several neighbor cells, leading to an index size increase and therefore to an efficiency degradation.

An example of a fixed grid built on a collection of 15 rectangles is shown in Figure 6.5, whereas Figure 6.6 shows a grid file for the same collection. In both cases, the page capacity is 4.

The insertion and deletion of objects is the same as for points. As a consequence of object duplication in neighbor cells (see, for example, object 6 in Figure 6.5), cell split is likely to occur more often. Figure 6.7 illustrates a first case of cell split. Starting with the grid file of Figure 6.6, the insertion of object 14 in cell $DIR[1, 3]$ triggers the overflow of page $p5$. A new boundary is created at value $x3$ in order to divide cell $DIR[1, 3]$ along the x axis. $x3$ has been inserted in S_x, and the directory now consists of $3x(2 + 1) = 9$ cells. Cells $DIR[2, 1]$ and $DIR[2, 2]$ reference existing pages (respectively, $p1$ and $p3$), each being

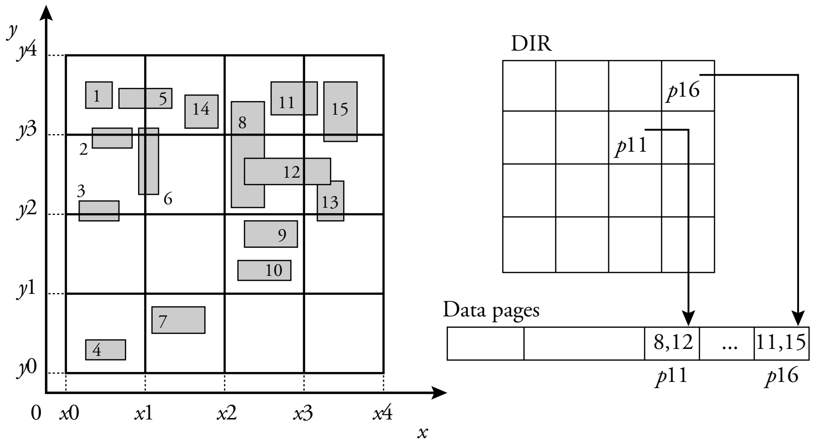


Figure 6.5 A fixed grid for rectangle indexing.

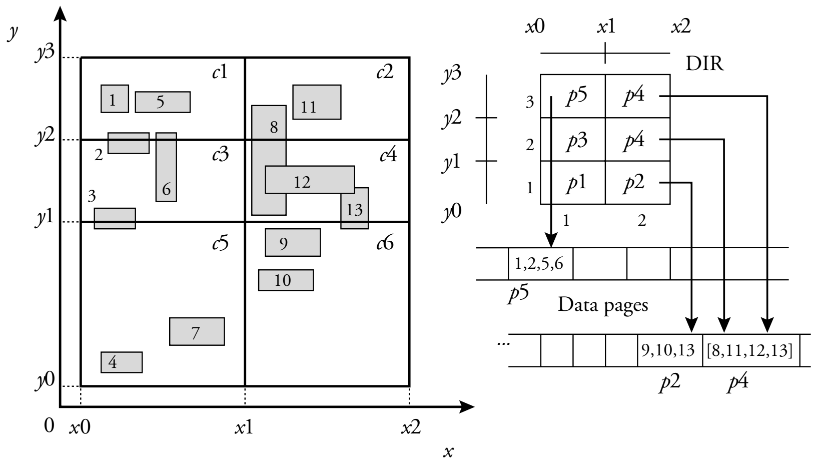


Figure 6.6 A grid file for rectangle indexing.

now assigned to two cells. The objects in *p*5 have been distributed among pages *p*5 and *p*6, respectively referenced by cells *DIR*[1, 3] and *DIR*[2, 3]. Observe the duplication of objects 5 and 6.

A cell split without directory split is shown in Figure 6.8: when inserting object 15, page *p*4 overflows. The new page *p*7 is allocated, and the cells *DIR*[3, 2] and *DIR*[3, 3] are updated in the directory.

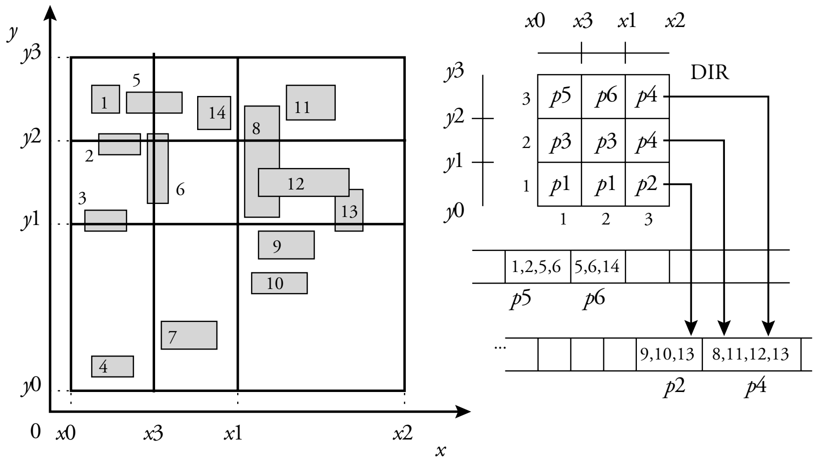


Figure 6.7 Insertion of object 14.

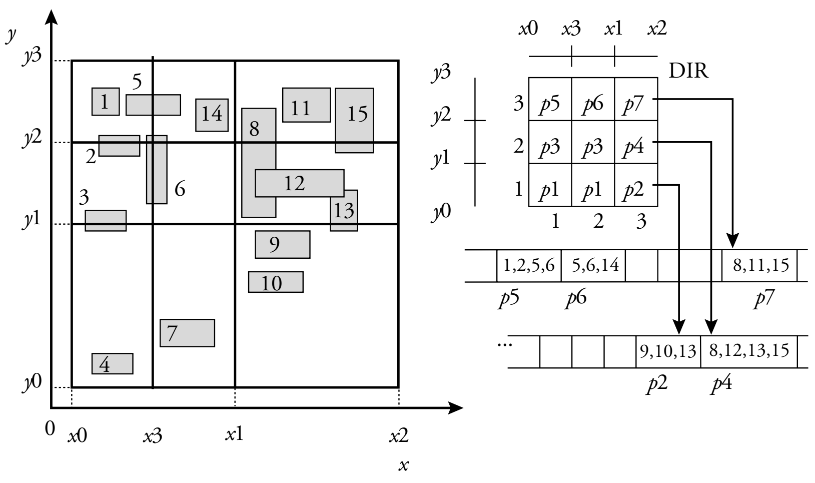


Figure 6.8 Insertion of object 15.

POINT AND WINDOW QUERIES

The point query algorithm works as follows. Given the point coordinates $P(a, b)$, one looks for the single cell containing P. In the case of the fixed grid, this can be done in constant (CPU) time, and in logarithmic time with the grid file by using a dichotomic search on the scales. One disk access is necessary to read the page referenced by the cell. One obtains a collection of entries[25] E. It remains, for each entry e in E, to test whether $e.mbb$ contains P. In a third step (refinement step), the object with identifier $e.oid$ is fetched in order to test for the inclusion of P in the actual geometry of the spatial object (see Chapter 5).

The algorithm follows. It returns the set of object identifiers for objects whose bounding box contains the point argument P. RANK(v, S) returns an integer, which is the rank of the interval in array S containing v (this can be implemented as a linear search or as a dichotomic search) whereas READPAGE(p) brings a page to main memory.

25. Recall that an index entry is a pair (mbb, oid), where mbb is the bounding box of an object with identifier oid.

ALGORITHM GF-POINTQUERY (*P(a, b):* point): set(*oid*)

```
begin
   result = ∅
   // Compute the cell containing P
   i = RANK(a, $S_x$); j = RANK(b, $S_y$)
   page = READPAGE (DIR[i,j].PageID)
   for each e in page do
      if (P ∈ e.mbb) then result += {e.oid}
   end for
   return result
end
```

An example of a point query on the grid file is shown in Figure 6.9. Cell *DIR*[3, 2] is retrieved, and page *p*4, containing four entries, is loaded. The sequential test on the four entries yields entries 8 and 12 containing *P*.

The window query algorithm works as follows. The first step consists of computing all cells that overlap the window argument. In a second step, each of them is scanned, as in the point query algorithm. Because the result is likely to have duplicates, they need to be removed. This is performed by sorting.

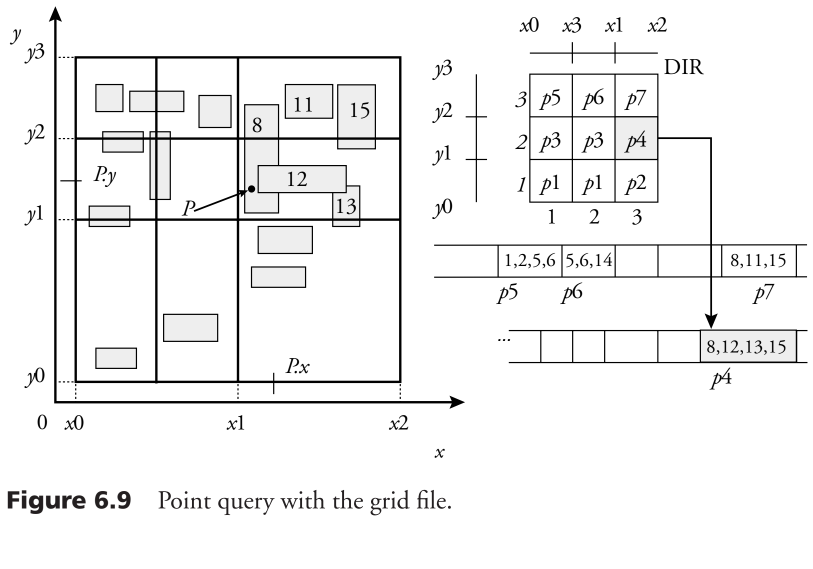


Figure 6.9 Point query with the grid file.

ALGORITHM GF-WINDOWQUERY ($W(x_1, y_1, x_2, y_2)$: rectangle): set(oid)

```
begin
  result = ∅
  // Compute the low-left cell intersecting W
  $i_1$ = RANK ($x_1, S_x$); $j_1$ = RANK ($y_1, S_y$)
  // Compute the top-right cell intersecting W
  $i_2$ = RANK ($x_2, S_x$); $j_2$ = RANK ($y_2, S_y$)
  // Scan the grid cells
  for ($i = i_1$; $i \leq i_2$; i++) do
    for ($j = j_1$; $j \leq j_2$; j++) do
      // Read the page, and test each entry
      page = READPAGE (DIR[i,j].PageID)
      for each (e in page) do
        if ($W \cap e.mbb \neq \emptyset$) then result += {e.oid}
      end for
    end for
  end for
  // Sort the result, and remove duplicates
  SORT (result); REMOVEDUPL (result);
  return result
end
```

Two points are noteworthy:

- Removing duplicates can be costly and space consuming. Its time complexity is superlinear in the size of the result.
- Loading several pages can be accelerated if they are consecutive on the disk (a sequential scan of n pages is far more efficient than n random accesses).

It is easy to see that a point query can be carried out in constant I/O time (the two disk accesses principle holds), whereas the number of I/Os for the window query is proportional to the window area.

DISCUSSION

The grid structure for rectangle indexing seems to meet most of the requirements discussed in Section 6.1. Nevertheless, several problems remain. First, the object duplication in neighbor cells increases the number of entries in the index, and therefore the index size. This effect becomes more and more significant as the data set size increases and as the cell size decreases down to the *mbb*'s size. Second, removing duplicates from the result is expensive when the result size is large. Last, but not least, the efficiency of the SAM relies on the assumption that the directory is resident in central memory. For very large data sets, experiments show that this is impossible. If the directory has to be partly on disk, its management becomes complex and the response time is degraded, in that supplementary I/Os are necessary to access the directory. There have been some proposals to improve this situation, which render the SAM much more complex.

6.2.2 The Linear Quadtree

The SAMs presented in this section and in the next enable a simple extension of DBMSs using their B-tree as an index. Rectangles to be indexed are mapped to cells obtained by a recursive decomposition of the space into quadrants, known as *quadtree decomposition.* Cells (and associated rectangles) are indexed by a B-tree using the cell rank as a key. The scheme presented in this section and called *linear quadtree* relies on the *mbb* of spatial objects, whereas the structure (called a *z-ordering tree*) presented in the following section approximates the object geometry not by its *mbb* but by cells from its quadtree decomposition. We first describe the *quadtree* on which all SAMs are based.

THE QUADTREE

This main memory data structure is commonly utilized for accelerating the access to objects in the 2D plane because of its simplicity. In the following we sketch a variant for indexing a collection of rectangles (the *mbb* of objects) stored on disk. The search space is recursively decomposed into quadrants until the number of rectangles overlapping each quadrant is less than the page capacity. The quadrants are named North West (NW), North East (NE), South West (SW), and South East (SE). The index is represented as a quaternary tree (each internal node has four children, one per quadrant). With each leaf is associated a disk page, which stores the index entries, as defined in Section 6.2.1. As in the case of the grid file, a rectangle appears in as many leaf quadrants as it overlaps. Figure 6.10 shows a partitioning of the search collection for the collection of Figure 6.7 and its associated tree. The leaves are labeled with their corresponding rectangles. We assume in this example a page capacity of four entries.

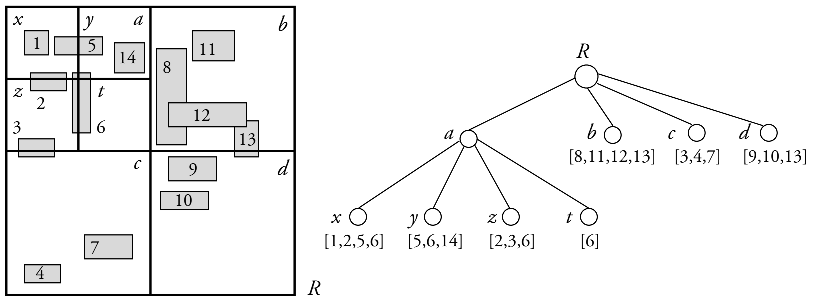


Figure 6.10 A quadtree.

Point query is simple with the quadtree. A single path is followed from the tree root to a leaf. At each level, one chooses among the four quadrants the one that contains the point argument. The leaf is read and scanned, as for the grid. Figure 6.11 illustrates a point query and the path followed from the root to a leaf, with rectangle 5 as a result. For the window query, one has to find all leaf quadrants that intersect the window argument. Because this SAM is not central to our study, we will not detail the algorithm, which is a bit more complex than for the grid. However, we shortly describe the dynamic insertion of rectangles.

A rectangle will be inserted in every leaf quadrant it overlaps. To insert the rectangle, as many paths as there are leaves that overlap the rectangle will be followed. The page p associated with each leaf is read. Then two cases may happen: either p is not full (a new entry is inserted) or it is full. Then the quadrant is subdivided into four quadrants. Three new pages are allocated. The entries of the old page as well as the new one are distributed among the four pages. An entry e is added to each

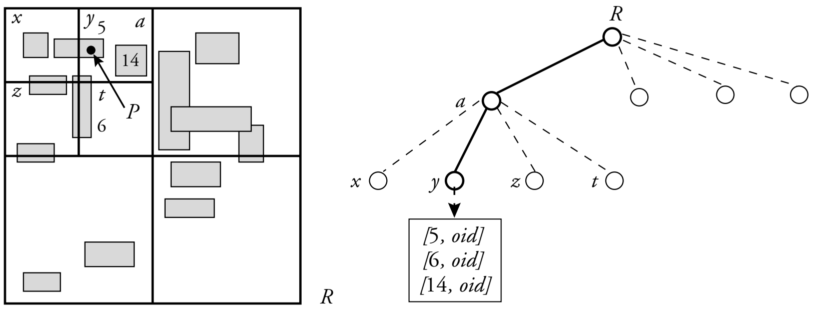


Figure 6.11 A point query on a quadtree.

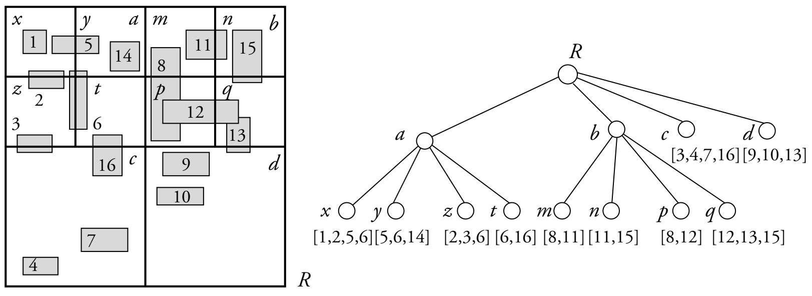


Figure 6.12 Insertion in a quadtree.

of the four pages, whose quadrants intersect $e.mbb$. Figure 6.12 shows the index obtained after the insertion of rectangles 15 and 16. The insertion of 15 leads to the split of *b* into quadrants *m, n, p*, and *q*. 15 is added to leaves *n* and *q*. The insertion of 16 does not lead to any split. 16 is added to leaves *c* and *t*.

This variant of a quadtree does not meet several of the requirements of a SAM. The essential reason for this mismatch is the small number of children (the node *fan-out*) fixed to 4, which occupies only a small part of a page. Then it is not easy to map a quadtree to disk pages. Tree structures with large node fan-out (such as the B-tree or R-tree, discussed in the following) allow one to efficiently map a node to a disk page and are thus more appropriate for secondary memory access methods.

The quadtree query time is related to the tree depth, which might be large. In the worst case, each internal tree node is in a separate page, and the number of I/Os is equal to the tree depth. If the collection is static, there exist efficient packings of the tree nodes onto pages, but the performance degrades in the dynamic case. Furthermore, like the grid file, the quadtree previously described suffers from object duplication in several leaves. When the collection size is so large that the quadrant size is of the order of the size of the rectangles to be indexed, the duplication rate (and therefore the index size) is exponentially increasing, and the SAM is not effective anymore.

SPACE-FILLING CURVES

A *space-filling curve* defines a total order on the cells of a 2D grid. This order is useful because it partially preserves proximity; that is, two cells close in space are likely to be close in the total order. Even though this is not always the case, some curves provide a reasonable approximation of this proximity property. The order should also be stable with respect to the resolution of the grid.

We describe two popular space-filling curves using, as an underlying grid, an $N \times N$ array of cells where $N = 2^d$. The grid can be seen as a complete quadtree with depth d.

The first variety (see Figure 6.13a), known as *z-order* or *z-ordering*, is generated as follows. A label is associated with each node of the complete quadtree, chosen among strings over the alphabet (0, 1, 2, 3). The root has for a label the empty string. The NW (respectively, NE, SW, SE) child of an internal node with label k has for a label $k.0$ (respectively, $k.1$, $k.2$, $k.3$), where "." denotes string concatenation. Then the cells are labeled with strings of size d. We can sort the cells according to their labels (in lexicographic order). For example, choosing a depth $d = 3$ and ascendent order, cell 212 is before cell 300 and after cell 21. The ordering NW, NE, SW, SE justifies its *z-order* name.[26] This order is also called Morton code.

The Hilbert curve is also a one-dimensional curve, which visits every cell within a 2D grid. The shape is not a Z but a Π. Unlike *z*-ordering,

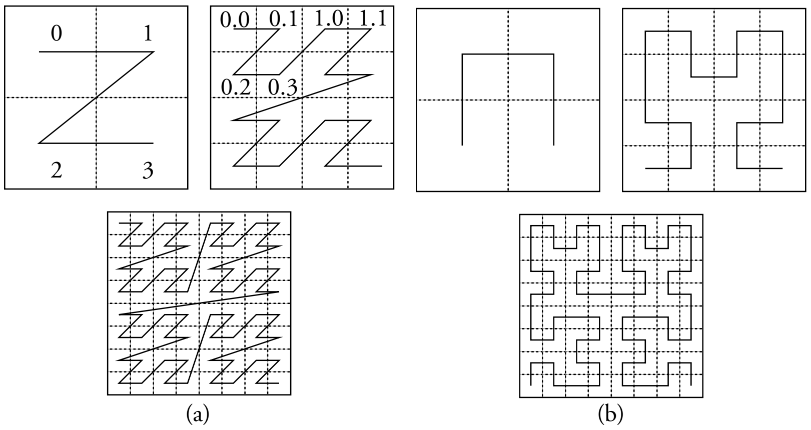


Figure 6.13 Space-filling curves: *z*-ordering (a) and Hilbert curve (b).

26. As a matter of fact, the original *z*-order (see bibliographic notes) has an "N" shape: the order between quadrants is SW, NW, SE, NE.

the Hilbert curve consists of segments of uniform length; that is, we never have to "jump" to a distant location while scanning the cells (Figure 6.13b). It is easy to see that in both cases there exist some unavoidable situations in which two objects are close in the 2D space, but far from one another on the space-filling curve.

QUADTREE LABELING

Now take a quadtree of depth d. It can be embedded in a grid with $N = 2^d$, as previously described. Any leaf quadrant can be labeled with a string of size $\leq d$. Figure 6.14 shows an example of such a quadtree. Observe (1) that the order of the leaves corresponds to a left-to-right scan of the leaves, and (2) that the labels are not of the same size. The size of the label is the depth of the leaf in the tree. The label of a leaf can also be seen as the label of a path from the root to the leaf.

Note also that if a leaf L contains a grid cell C, then the label l of L is a *prefix* of the label c of C, and we have $l < c$, where the order $<$ is the ascendent lexicographic order on strings. For example, $3 < 31 < 312$.

LINEAR QUADTREE

The key observation is that once the entries $[mbb, oid]$ have been assigned (as for a regular quadtree) to a quadtree leaf with label l, and stored in a page with address p, then we index in a B+tree the collection of pairs (l, p) keyed on the leaf label l (Figure 6.15).

Such a structure provides a nice packing of quadtree labels into B+ tree leaves. The packing is dynamic; that is, it persists when inserting and deleting objects in the collection. For example, entries corre-

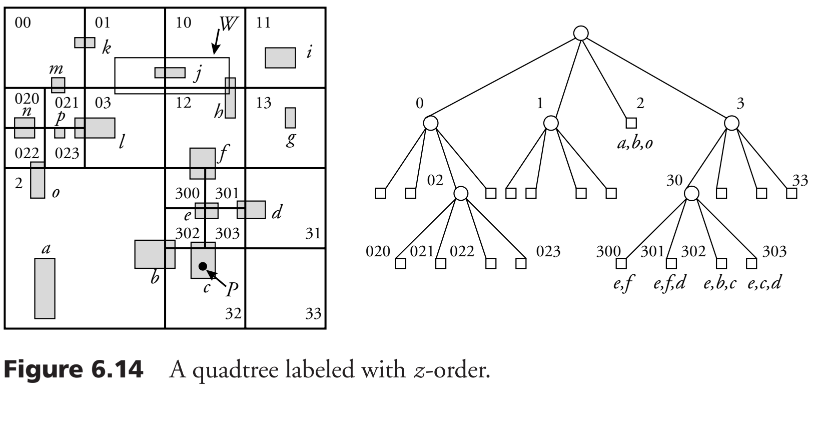


Figure 6.14 A quadtree labeled with z-order.

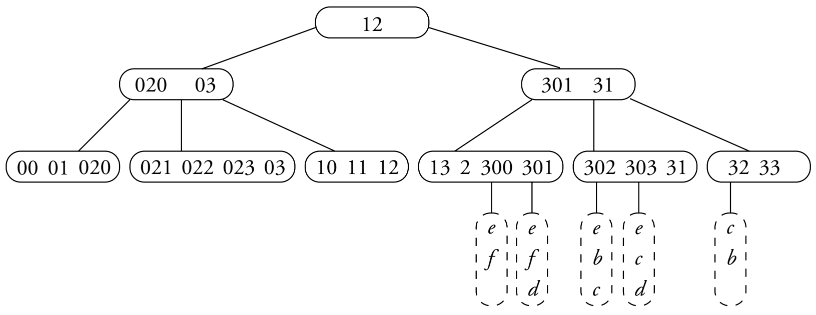


Figure 6.15 The leaves of the quadtree indexed with a B+tree.

sponding to leaves with labels 13, 2, 300, and 301 are in the same page. However, such a scheme has the same redundancy problem as the variant of the quadtree previously presented: *mbb*s that overlap several quadtree leaves are duplicated in pages associated with these leaves. As an example, *e* is stored in four pages. We will see in the following another linear structure that avoids duplication of *mbb*s.

POINT QUERY WITH A LINEAR QUADTREE

Retrieving identifiers of objects whose *mbb* contains a point *P* necessitates three steps:

- *Step 1: point label computation.* Compute the label l (a string of length *d*) of the grid cell containing *P*. This is performed by function POINTLABEL.
- *Step 2: quadtree leaf retrieval in the B+tree.* Let L be the label of the quadtree leaf that contains *P*. Then either $L = l$, the quadtree leaf is at depth d, or L is a prefix of l: its length is less than d and the leaf quadrant contains the grid cell including *P*. More precisely, this quadrant corresponds to the entry in the B+tree whose key L is the largest value less than or equal to l. Looking for such an entry is not a basic function provided with B+trees. We call such a function MAXINF (l). Although simple, it requires a slight modification of the code usually associated with this access method.
- *Step 3: leaf access and scan.* Once the B+tree entry [*L*, *p*] has been found, one must as usual access the page with address *p*, scan all pairs [mbb, oid] in this page, and check which *mbb*(s) contains point *P*. The result is a list of object identifiers.

This algorithm is illustrated in Figure 6.14. From the coordinates of *P*, one finds that the grid cell containing *P* has for a label 320 (first step). One then retrieves in the B+tree, using function MaxInf, the maximal leaf label smaller than 320; that is, the leaf with label 32 (step 2). It remains (step 3) to access the page and scan the entries checking for each of them whether the *mbb* contains *P*: object *c* is found. The algorithm is summarized in the following.

ALGORITHM LQ-PointQuery (*P:* point): set(*oid*)

```
begin
   result = ∅
   // Step 1: compute the label of the point
   l = PointLabel(P)
   // Step 2: the entry [L, p] is obtained by traversing the B+tree with key l.
   [L, p] = MaxInf (l)
   // Step 3: get the page and retrieve the objects
   page = ReadPage (p)
   for each e in page do
      if (e.mbb contains P) then result += {e.oid}
   end for
   return result
end
```

WINDOW QUERY WITH A LINEAR QUADTREE

The idea is to compute the interval *I* of labels covering all quadrants that overlap window *W*, and then to issue a range query on the B+tree with *I*. This is done in three steps.

- *Step 1.* Compute the label *l* of the NW window vertex, as in step 1 of the point query algorithm. Then compute MaxInf(*l*), as in step 2 of that algorithm. This yields [*L*, *p*], where label *L* is the lower bound of the interval. Then compute the label *l′* of the SE window vertex and MaxInf (*l′*). This gives [*L′*, *p′*], where *L′* is the upper bound.
- *Step 2.* Issue a range query with interval [*L*, *L′*] on the B+tree, which retrieves all entries [*l*, *p*] whose label *l* lies in this interval.
- *Step 3.* For each B+tree entry *e* = [*l*, *p*], compute the quadrant labeled by *e.l* with function Quadrant(*e.l*). If Quadrant(*e.l*) over-

laps the window, access the quadtree page $e.p$ and test each of the quadtree entries $[mbb, oid]$ for overlap with W.

Consider again the example of Figure 6.14 and the query window with argument W. The label of the NW vertex of W is 012, and the label of the SE vertex is 121. One then searches the B+tree of Figure 6.15 with function MAXINF for the largest quadtree labels less than 012 and 121, respectively. One obtains MAXINF(012) = 01 and MAXINF(121) = 12 (end of step 1).

Then (step 2) a range query on the B+tree of Figure 6.14 with interval [01, 12] is issued. For each entry e found during the scan (step 3), we access the page *only* if W overlaps QUADRANT($e.l$). This is not necessary, for instance, for the entries with labels 020–023 and 11. The algorithm is summarized in the following.

ALGORITHM LQ-WINDOWQUERY (W: rectangle): set(*oid*)

```
begin
  result = ∅
  // Step 1: From the vertices W.nw and W.se of the window, compute
  // the interval [L, L']. This necessitates two searches through the B+tree.
  l = POINTLABEL (W.nw); [L, p] = MAXINF (l)
  l' = POINTLABEL (W.se); [L', p'] =MAXINF(l')
  // Step 2: The set Q of B+tree entries [l, p] with l ∈ [L, L'] is computed.
  Q = RANGEQUERY ([L, L'])
  // Step 3: For each entry in Q whose quadrant overlaps W, access
  // the page
  for each q in Q do
    if (QUADRANT (q.l) overlaps W) then
      page = READPAGE (q.p)
      // Scan the quadtree page
      for each e in page do
        if (e.mbb overlaps W) then result += {e.oid}
      end for
    end if
  end for
  // Sort the result, and remove duplicates
  SORT (result); REMOVEDUPL (result);
  return result
end
```

The price to pay for mapping a 2D collection of quadrants onto a one-dimensional list of ordered cells is, as expected, an important clustering mismatch. In the interval $[L, L']$, there may be a large number of quadrants that do not overlap window W. Therefore, the range query might require more I/O than necessary. However, the pages associated with the quadrants that do not overlap W are not accessed.

INSERTION OF RECTANGLES

Two cases must be considered: either there is no split of the embedded quadtree, and the quadrant page is accessed and updated, or there is a split of the embedded quadtree. In this case, one entry of the B+tree must be deleted and replaced by four new entries (i.e., one per new quadtree page).

ANALYSIS

The number of I/Os for a point query has two components: (1) the number of I/Os for accessing a leaf of the B+tree (step 2), and (2) access to the quadrant page (step 3). Then the number of I/Os is $d + 1$, where d is the depth of the B+tree. We know that this is true even with skewed distributions, or in dynamic situations, because of the B+tree structure whose space and time performance is dependent on its size only. This efficient behavior, even in worst-case conditions, is at the origin of its use in the design of such a SAM.

The number of I/Os for a window query is more difficult to evaluate. Step 1 requires two accesses to the B+tree (i.e., $2d$ I/Os). Step 2 requires d I/O to reach the leaf corresponding to the lower bound of the interval, and then as many I/Os as there are chained B-tree leaves to be scanned. The number of I/Os of step 2 is dependent on the size of the interval (i.e., the size of the window). There exist more efficient algorithms, which avoid searching for quadrants that do not overlap the window argument, but the price to be paid is a serious increase in design complexity.

6.2.3 *The z-Ordering Tree*

In contrast to all SAMs studied in this chapter, the structure detailed in the present section does not use as an approximation of the object its *mbb*. Instead, the geometry of each object is decomposed into a quadtree of depth bounded by d, and one indexes the set of quadtree leaves that approximate the geometry. The leaves' labels (whose size is $\leq d$) are then inserted into a B+tree. The structure is known as a *z-ordering tree.*

The construction algorithm uses the following basic step. Given an object geometry o and a quadrant q, one checks whether o overlaps *all* leaves of the complete quadtree rooted at q. Then q is reported as an element of the resultant set. Otherwise, one decomposes o into as many pieces (≤ 4) as there are subquadrants of q overlapping o, and recursively applies the same process to each piece. The recursion stops when the maximal depth d is reached. The quadrant is then said to be *minimal.*

Alternately, the resulting set can be seen as a raster approximation of the object with a fixed grid of 4^d cells, in which sibling (NW, NE, SW, SE) cells, if any, have been recursively compounded.

Figure 6.16 illustrates the object decomposition and approximation. The object approximation is the list of quadrants with labels {023, 03, 103, 12, 201, 210, 211, 300, 301, 302}. All are minimal quadrants (or grid cells) at depth 3, except two quadrants at depth 2 with labels 03 and 12. The following algorithm summarizes the decomposition process. The operator $+$ denotes set union, whereas $\cup$ denotes geometric union. At the beginning, Decompose is called with q = search space.

ALGORITHM **Decompose** (o: geometry, q: quadrant): set(quadrant)

```
begin
  decomp_NW, decomp_NE, decomp_SW, decomp_SE: set(quadrant)
  result = Ø
  // Check that q overlaps o, else do nothing
  if (q overlaps o) then
    if (q is minimal) then
      result = {q}
    else
      // Decompose o into pieces, one per subquadrant
      for each sq in {NW(q), NE(q), SW(q), SE(q)} do
        decomp_sq = DECOMPOSE (o, sq)
      end for
      // If each decomposition results in the full subquadrant, return q
```

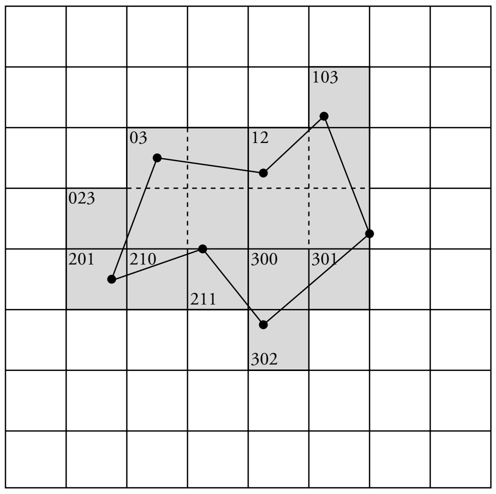
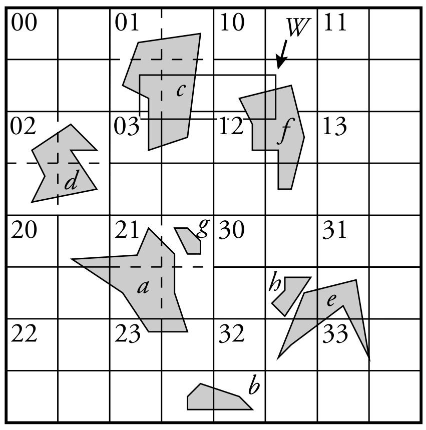


Figure 6.16 *z*-ordering and object decomposition.

```
            if ($decomp_{NW} \cup decomp_{NE} \cup decomp_{SW} \cup decomp_{SE} = q$) then
                $result = \{q\}$
            else
                // Take the set union of the four decompositions
                $result = decomp_{NW} + decomp_{NE} + decomp_{SW} + decomp_{SE}$
            end if
        end if
    end if
    return result
end
```

A collection of eight objects, together with their *z*-ordering decomposition, is shown in Figure 6.17. The quadtree has a maximal depth



a = {201, 203, 21, 230, 231}

b = {233, 322}

c = {01, 030, 031}

d = {02}

e = {303, 312, 321, 330}

f = {102, 103, 120, 121, 123}

g = {211}

h = {303}

Figure 6.17 A set of objects with *z*-ordering decomposition.

$d = 3$. We obtain a set of entries $[l, oid]$, where l is a cell label and *oid* is the identifier of the object whose approximation contains or overlaps the cell with label l. For instance, object *a* in Figure 6.17 is represented by the following entries:

{[201, *a*], [203, *a*], [21, *a*], [230, *a*], [231, *a*]}

Obviously, this scheme, like the previous linear quadtree, implies duplication. An object's *id* is in as many entries as there are cells in its approximation. Conversely, there may be in the resulting collection several entries with the same l but with different *id*s when several object approximations share the same cell at the same level. This is the case for objects *e* and *h*, which share label 303. Finally, note that some objects may overlap the same cell, but at different decomposition levels (see objects *a* and *g*).

Entries are then inserted in a B+tree keyed on their label l. Figure 6.18 shows the B+tree obtained from the *z*-ordered data set of Figure 6.17. The *id*s are available in the B+tree leaf pages, whereas with the linear quadtree (see Figure 6.15), a supplementary page access from the B+tree leaf page is necessary. It is worth noting that the same object may be represented in two quite distant leaves of the B+tree, because of some unavoidable "jump" in the *z*-order. For example, object *b* is decomposed into two cells: the corresponding entries are stored in two distinct and non-neighbor pages of the B+tree.

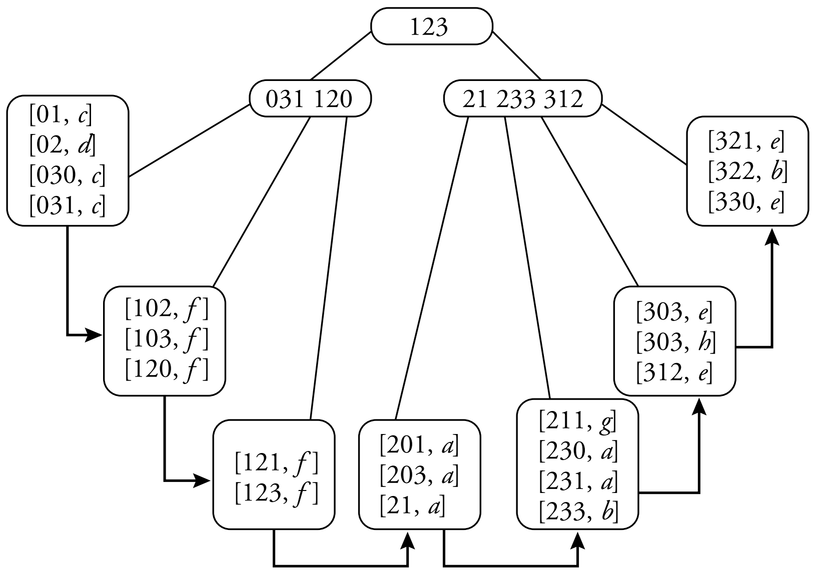


Figure 6.18 *z*-ordering tree.

The point and window algorithms are very similar to those for the linear quadtree. In the last step, one only needs to keep the *oid* of entries that overlap the window argument. The following is the WINDOWQUERY algorithm. The point query algorithm is left as an exercise.

ALGORITHM ZO-WINDOWQUERY (*W*: rectangle): set(*oid*)

```
begin
  result = ∅
  // Step 1: From the vertices W.nw and W.se of the window, compute
  // the interval [L, L']. This necessitates two searches through the B+tree.
  l = POINTLABEL (W.nw); [L, p] = MAXINF (l)
  l' = POINTLABEL (W.se); [L', p'] = MAXINF (l')
  // Step 2: Compute the set E of entries [l, oid] such that l ∈ [L, L'].
  E = RANGEQUERY ([L, L'])
  // Step 3: For each entry in E that overlaps W, report the oid
  for each e in E do
    if (QUADRANT (e.l) overlaps W) then result += {e.oid}
  end for
```

```
   // Sort the result, and remove duplicates
   SORT (result); REMOVEDUPL (result);
   return result
end
```

Examine window W in Figure 6.17. The labels of the minimal quadrants that contain $W.nw$ and $W.se$ are, respectively, 012 and 121. Function MAXINF gives, respectively, the entries [01, *c*] and [121, *f*], and the range query on the B+tree scans the first three leaves of the structure. Some entries are scanned, such as [02, *d*], which are not relevant to the query.

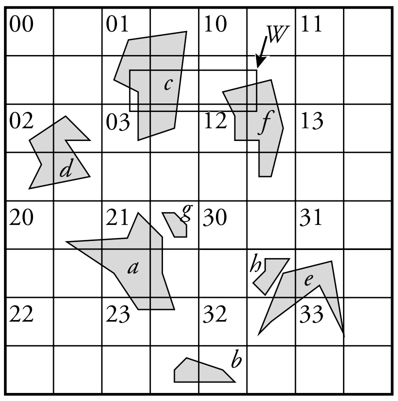
The number of I/Os for a window query depends on the window size. Step 1 requires one to access the B+tree twice ($2d$ I/Os, where d denotes the depth of the B+tree). Step 2 requires d I/Os to reach the leaf corresponding to the lower bound of the query interval, and then as many I/Os as the number of chained B+tree leaves to be scanned.

It is interesting to compare the linear quadtree to the *z*-ordering tree. With the latter structure, the B-tree depth, and therefore the number of I/Os, is likely to be larger because there are as many entries as there are cells in the decomposition of all objects. The finer the grid resolution, the better the objects' approximation but the larger the number of entries to be indexed in the B+tree.

With the linear quadtree, the number of B+tree entries is much smaller, because it is equal to the number of quadtree quadrants. However, a supplementary I/O is required per quadrant overlapping the window.

A RASTER VARIANT OF *z*-ORDERING (WITH REDUNDANCY)

A variant of the previous scheme consists of decomposing the object into elementary grid cells. The approximation is the set of minimal grid cells that contain, are contained by, or overlap the object. In other words, there is no gathering of sibling cells into larger quadrants, and the resulting set of cells is the *raster* representation of the object. This is shown in Figure 6.19 on the collection of objects of Figure 6.17. Object *d*, for instance, is now represented by four cells instead of one (see the difference on object *a* in Figure 6.17, for instance). All labels have the same size, equal to the depth $d = 3$.



a = {201, 203, 210, 211, 212, 213, 230, 231}
b = {233, 322}
c = {010, 011, 012, 013, 030, 031}
d = {020, 021, 022, 023}
e = {303, 312, 321, 330}
f = {102, 103, 120, 121, 123}
g = {211}
h = {303}

Figure 6.19 A set of objects with raster decomposition.

In this particular case, the redundancy is worse, and the number of entries in the B+tree is larger. However, the algorithms for point and window queries (as well as spatial joins, Chapter 7) are much simpler. In particular, for the window query, the interval bounds are grid cell labels computed with function POINTLABEL. Because all cells are minimal, we do not need function MAXINF and therefore can use the B+tree existing code implemented in each DBMS without any modification. Figure 6.20 shows a B+tree for the raster decomposition. Given the same window *W*, with point labels 012 and 121, we can now directly issue a range query on the interval [012, 121]. Indeed, any object *o* overlapping the cell with label 012 is represented by an entry [012, *o*]. The WINDOWQUERY algorithm follows.

ALGORITHM **ZR-WINDOWQUERY** (*W*: rectangle): set(*oid*)

```
begin
  result = ∅
  // Step 1: From the vertices W.nw and W.se of the window, compute
  // the interval [L, L′]. This necessitates two searches through the B+tree.
  L = POINTLABEL (W.nw); L′ = POINTLABEL (W.se)
  // Step 2: Compute the set E of entries [l, oid] such that l ∈ [L, L′].
  E = RANGEQUERY ([L, L′])
  // Step 3: For each entry in E that overlaps W, report the oid
  for each e in E do
    if (QUADRANT (e.l) overlaps W) then result += {e.oid }
```

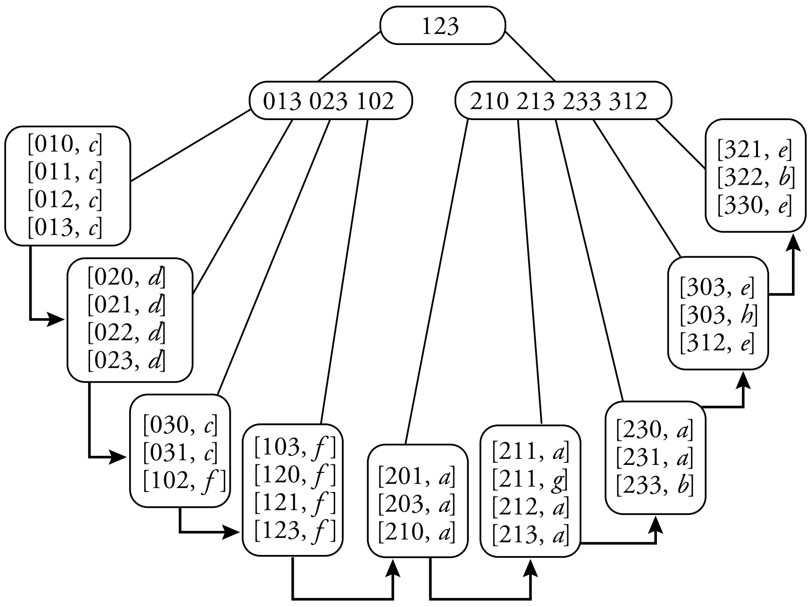


Figure 6.20 *z*-raster tree.

```
  end for
  // Sort the result, and remove duplicates
  SORT (result); REMOVEDUPL (result);
  return result
end
```

z-ORDERING WITHOUT REDUNDANCY

We end the presentation of linear SAMs with a variant of the *z*-ordering scheme without redundancy. The idea is the following. An object is not assigned to all quadrants it overlaps, but to the *smallest* quadrant that contains it.

In Figure 6.21, *b* is assigned to the initial search space with an empty label (_). Objects *c, d*, and *g* are, respectively, assigned to quadrants with labels 0, 02, and 211. The advantage of this SAM is that there are no duplications of objects into several quadrants. A drawback, however, is that the ratio between the object size and the assigned quadrant size might be small. In the case, such as for *b*, where the quadrant is close to the root, this implies that the object appears—as a false drop—in a

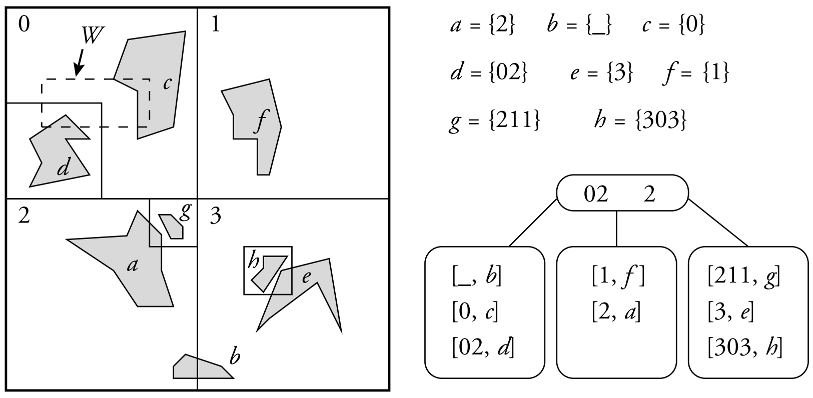


Figure 6.21 Z-ordering without redundancy.

large number of queries. Whatever the position of the point or window query, *b* is part of the candidate set of *all* queries, and thus will be processed during the refinement step.

The structure itself is a simple variant of the *z*-ordering tree. With each quadtree node, whether internal or leaf, is associated a list of objects contained in this quadrant. Each quadrant is assigned a label, as previously defined, which is a string over {0, 1, 2, 3} or the empty string (_) for the root. For each indexed object, there is an entry [l, *oid*], where l is the label of the quadrant the object has been assigned to, and *oid* is the *id* of the object. As for the *z*-ordering tree, the entries are stored in a B+tree keyed on the label (see Figure 6.21). In contrast to the *z*-ordering scheme, there is a single entry per object. Thus, the B+tree is significantly less deep.

The following is an algorithm for the window query. We leave as an exercise the algorithm for a point query. The algorithm requires a function MINSUP(v), which looks for the smallest entry in the B+tree whose key is larger than or equal to v. PREFIX(k) is the set of labels that are prefix strings of k, including k. As an example, PREFIX(0312) yields {_, 0, 03, 031, 0312}. NEXT(k) is the next entry in sequence in the B+tree leaf (or in the next chained B+tree leaf), after the entry k. The window argument W is also assigned the smallest quadtree quadrant containing W, with label $W.l$.

ALGORITHM **ZN-WindowQuery** (*W:* rectangle): set(*oid*)

```
begin
  result = ∅
  // Access the B-tree for each prefix of W.l
  for each v in Prefix(W.l) do
    // k is the first entry whose label is greater than or = v
    k = MinSup(v)
    // Scan the entries with label v if any, or (only for the last B-tree
    // access: v = W.l), the entries whose key is a suffix of W.l
    while (k.l = v) or (v = W.l and W.l in Prefix(k.l)) do
      // Keep the oid of an entry
      result +={k.oid}
      k =Next(k) // Next entry in sequence
    end while
  end for
  return result
end
```

The B+tree is accessed $n + 1$ times if n is the size of the label $W.l$ of the window argument. For each prefix v of $W.l$ (each of the first n times), one accesses the B-tree and looks for the smallest entry k whose label is larger or equal to v. It might happen that the label of k is larger than v. Then we go to the next v. Otherwise ($k.l = v$), one keeps the entry *oid* and goes to the next entry in sequence. During the last B-tree access (quadrant with label $W.l$), one takes not only the objects assigned to the quadrant with label $W.l$, if any, but the objects assigned to the subquadrants (i.e., such that $W.l$ is a prefix of $k.l$).

In regard to the algorithm, examine Figure 6.21. The window argument is assigned to the quadrant with label 0. The first B+tree access with key value $v = '\_'$ yields entry [_, *b*]. The second, and last, access, with key value 0, gives [0, *c*], and scans the leaves of the B-tree until a label that is not a suffix of 0 is found. In this case, the scan stops at label 1. Entry *d* is found. The candidate set is {*b*, *c*, *d*}. In the refinement step, object geometries are accessed and checked against the window argument.

Note that the geometric representation of *b* will be accessed, although this is clearly unnecessary. Then many objects represented by small-size labels (close to the root of the quadtree) are likely to be accessed in a systematic way, whatever the query. Such a situation should deteriorate the query performance.

A simple optimization is to store in the entries of the B+tree the *mbb* of each object, in addition to its *oid*. Then a test with the *mbb* prior to the insertion of an object in the result set avoids a lot of false drops.

6.2.4 *Remarks on Linear SAM*

To conclude, by introducing a mapping from a two-dimensional partitioning of the plane into a one-dimensional list of cells, we enable the use of a B+tree, which is dynamic and efficient in terms of memory space and response time, even in worst-case conditions. More importantly, because commercial DBMSs use as an access method a B+tree, it is then easy to extend the system to handle spatial data. Indexing spatial data through the DBMS B+tree not only provides reasonably efficient access but allows one to benefit from other system features, such as concurrency control, recovery, access from the query language, and so on. Another advantage is that such a technique shown on a quadtree can be used with other space-partitioning SAMs.

There are several disadvantages, however. First, as discussed at length earlier on, the mapping of two-dimensional data sets onto a one-dimensional order leads to a clustering loss, which in turn leads to less efficient window queries. However, the main drawback of these structures, except for the last linear SAM, is the duplication of objects in neighbor cells. This increases the index size, decreases the access performance, and leads to a possibly expensive sort of the result for duplicate removal.

6.3 DATA-DRIVEN STRUCTURES: THE R-TREE

The SAMs of the R-tree family belong to the category of data-driven access methods. Their structure adapts itself to the rectangle distribution in the plane. Like the B-tree, they rely on a balanced hierarchical structure, in which each tree node, whether internal or leaf, is mapped onto a disk page. However, whereas B-trees are built on single-value keys and rely on a total order on these keys, R-trees organize rectangles according to a containment relationship. With each node is associated a rectangle, denoted the *directory rectangle* (*dr*) in the following, which is the minimal bounding box of the rectangles of its child nodes.

To access one rectangle of the indexed collection, one typically follows a path from the R-tree root down to one or several leaves, testing each directory rectangle at each level for either containment or overlap. Because, like the B-tree, the depth d of an R-tree is logarithmic in the number of objects indexed, and each node is mapped onto a disk page, the number of I/Os in nondegenerated situations is logarithmic in the collection size, and therefore the SAM is quite time and space efficient.

We first present the R-tree structure as originally proposed. Because in some situations the search time degrades, several variants have been proposed. Two of them, called R∗tree and R+tree, are described in subsequent sections.

6.3.1 The Original R-Tree

An R-tree is a depth-balanced tree in which each node corresponds to a disk page. A *leaf node* contains an array of *leaf entries*. A leaf entry is a pair (mbb, oid), with the usual meaning. mbb is of the form ($x_1, y_1, \ldots x_d, y_d$), where d is the search space dimension. Although the R-tree structure can handle data with arbitrary dimensions, we limit the presentation to the 2D case. A *non-leaf node* contains an array of node entries. The structure satisfies the following properties:

- For all nodes in the tree (except for the root), the number of entries is between m and M, where $m \in [0, M/2]$.
- For each entry (*dr*, *nodeid*) in a non-leaf node N, *dr* is the directory rectangle of a child node of N, whose page address is *nodeid*.
- For each leaf entry (*mbb*, *oid*), *mbb* is the minimal bounding box of the spatial component of the object stored at address *oid*.
- The root has at least two entries (unless it is a leaf).
- All leaves are at the same level.

Figure 6.22 shows an R-tree with $m = 2$ and $M = 4$. The indexed collection C contains 14 objects. The directory rectangles of leaves a, b, c, and d are represented by a dotted line.

The properties previously listed are preserved under any dynamic insertion or deletion. Observe that the structure, while keeping the tree balanced, adapts to the skewness of a data distribution. A region of

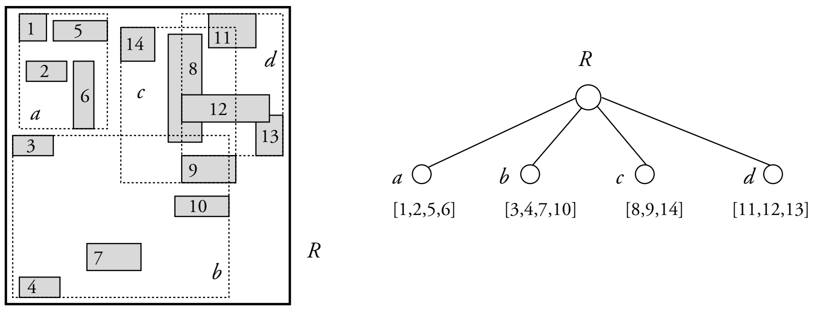


Figure 6.22 An R-tree.

the search space populated with a large number of objects generates a large number of neighbor tree leaves. This is not the case with space-partitioning trees, such as the quadtree. With the latter, some branches in the tree might be long, corresponding to regions with a high density of rectangles, whereas others might be short, the depth of the final tree corresponding to the region with highest density.

Figure 6.23b shows the directory rectangles of the leaves of an R-tree built on the hydrography data set of the state of Connecticut (Figure 6.23a). The rectangles' overlapping is obviously more important in dense areas than in sparse areas.

M, the maximal number of entries in a node, depends on the entry size $Size(E)$ and the disk page capacity $Size(P)$: $M = \lfloor Size(P)/Size(E) \rfloor$. Note that M might differ for leaf and non-leaf nodes,

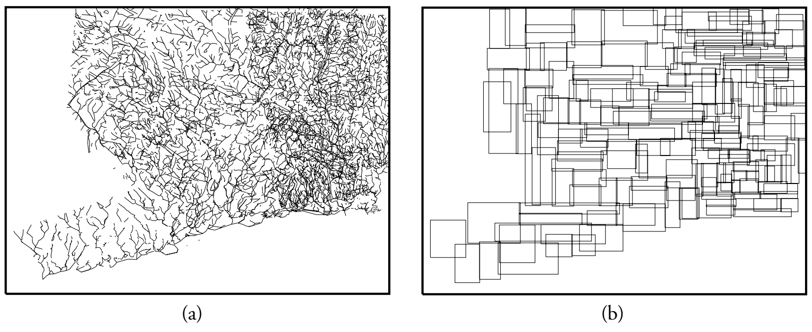


Figure 6.23 Indexing the hydrography network of Connecticut: TIGER data (simplified) (a) and R-tree (leaf level) (b).

because the size of an entry depends on the size of *nodeid* (non-leaf nodes) and *oid* (leaf nodes). *oid*, an object address inside a page, is generally longer than *nodeid*, which is a page address. The tuning of the minimum number of entries per node m between 0 and $M/2$ is related to the splitting strategies of nodes, discussed further in the following.

Given M and m, an R-tree of depth d indexes at least m^{d+1} objects and at most M^{d+1} objects. Conversely, the depth of an R-tree indexing a collection of N objects is at most $\lfloor \log_m(N) \rfloor - 1$ and at least $\lfloor \log_M(N) \rfloor - 1$. The exact value depends on the page utilization.[27]

To illustrate the R-tree time efficiency, assume the page size is 4K, the entry size is 20 bytes (16 bytes for the *mbb*, 4 bytes for the *oid*), and m is set to 40% of the page capacity. Hence, $M = 204$ and $m = 81$. An R-tree of depth 1 can index at least 6561 objects, whereas an R-tree of depth 2 can index at least 531,441, and up to 8,489,664, objects. Then, with a collection of one million objects, it takes three page accesses to traverse the tree, and a supplementary access to get the record storing the object. This example shows that even for large collections only a few page accesses are necessary to get down the tree in order to access the objects.

Unlike the space-partitioning SAMs seen previously, an object appears in one, and only one, of the tree leaves. However, the rectangles associated with the internal nodes do *not* constitute an exact partition of the space, and hence may overlap. In Figure 6.22, rectangles *b, c*, and *d* (which correspond to distinct leaves) overlap (see also Figure 6.23).

SEARCHING WITH R-TREES

This section presents a detailed algorithm for a point query. The algorithm for window queries is almost identical. The function RT-POINTQUERY is performed in two steps. First, one visits all children of the root whose directory rectangle contains the point P. Recall that because several directory rectangles may overlap, it might happen that the point argument falls into the intersection of several rectangles. All subtrees rooted at the intersecting rectangles then have to be visited.

27. The page utilization is defined as the ratio $[NbEntries \times Size(E)]/[NbPages \times Size(P)]$ between the space actually occupied by all entries (number of entries multiplied by the entry size) over the index space; that is, the number of pages (nodes) multiplied by the page capacity.

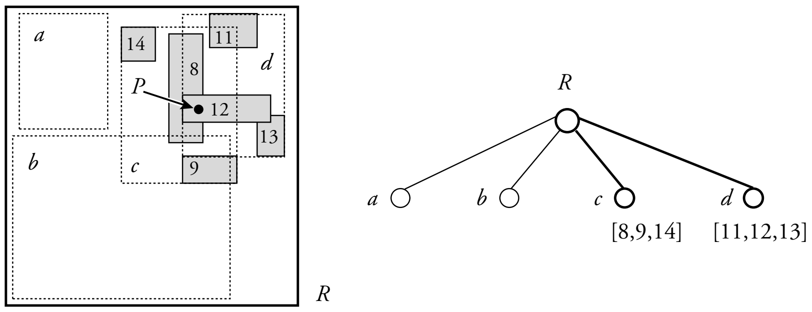


Figure 6.24 Point queries with R-trees.

The process is repeated at each level of the tree until the leaves are reached. For each visited node N, two situations may occur:

- At a given node, no entry rectangle contains the point, and the search terminates. This can occur even if P falls into the directory rectangle of the node. P falls into the so-called *dead space* of the node.
- P is contained by the directory rectangle of one or several entries. One must visit each associated subtree.

Hence, unlike the space-partitioning SAMs (i.e., the grid file or linear SAMs), several paths from the root to the leaves might be traversed. In a second step, the set of leaves produced in the first step is sequentially scanned: each leaf's entry whose *mbb* contains P is kept.

Figure 6.24 shows an example of a point query with P, which belongs to objects 8 and 12, as an argument. The query leads to the access of three nodes; namely, R, c, and d. A point query on the South-East corner point of object 8 would involve accessing four nodes; namely, R, b, c, and d.

The POINTQUERY algorithm with an R-tree follows.

ALGORITHM RT-POINTQUERY (P: point): set (*oid*)

```
begin
  result = ∅
  // Step 1: Traverse the tree from the root, and compute SL, the
  // set of leaves whose dr contains P
  SL = RTREETRAVERSAL (root, P)
```

```
   // Step 2: scan the leaves, and keep the entries that contain P
   for each L in SL do
      // Scan the entries in the leaf L
      for each e in L do
         if (e.mbb contains P) then result + = {e.oid}
      end for
   end for
   return result
end
```

The algorithm for traversing the tree is given by the function RTREE-TRAVERSAL (see the following algorithm). It implements a left and depth-first traversal of the R-tree through recursive calls. The function is initially called with the root page address as the first argument.

ALGORITHM RTREETRAVERSAL (*nodeid:* PageID, *P:* point): set of leaves

```
begin
   result = ∅
   // Get the node page
   N = READPAGE (nodeid)
   if (N is a leaf) return {N}
   else
      // Scan the entries of N, and visit those that contain P
      for each e in N do
         if (e.dr contains P) then
            result += RTREETRAVERSAL (e.nodeid, P)
         end if
      end for
   end if
   return result
end
```

It should be stressed that this algorithm is simplified from a memory management viewpoint. Keeping a set of leaves in memory would imply a large and unnecessary buffer pool. In an actual implementation, a one-page-at-a-time strategy should be used. We consider this issue in more detail in Chapter 7.

If the point falls into only one rectangle at each level, the point query necessitates *d* page accesses, where *d* is the depth of the tree, logarithmic in the number of indexed rectangles. Even if this seldom happens, one expects in practical situations that only a small number of paths are followed from root to leaves, and the expected number of I/Os is still logarithmic. Unfortunately, there are degenerated situations in which this number is not logarithmic anymore. In the worst case, it may occur that all directory rectangles of the leaves have a common intersection into which the point argument *P* falls. In this case, the entire tree must be scanned. However, despite its bad worst-case behavior, the R-tree is efficient in most practical situations.

The fact that the number of nodes visited is closely related to the overlapping of directory rectangles gave rise to numerous variants of the R-tree that intend to minimize this overlapping. You will see two of them at the end of the chapter.

The window query algorithm is a straightforward generalization of the point query in which the "contains *P*" predicate is replaced by the "intersect *W*" predicate, where *W* is the window argument. The larger the window, the larger the number of nodes to be visited.

INSERTION AND DELETION IN AN R-TREE

To insert an object, the tree is first traversed top-down, starting from the root (function INSERT). At each level, either one finds a node whose directory rectangle contains the object's *mbb* and then gets down the node subtree or there is no such node. Then a node is chosen such that the enlargement of its *dr* is minimal (function CHOOSESUBTREE). We repeat the process until a leaf is reached.

If the leaf is not full, a new entry [*mbb*, *oid*] is added to the page associated with the leaf. If the leaf directory rectangle has to be enlarged, the corresponding entry in the parent's node must be updated with the new value of *dr* (function ADJUSTENTRY). Note that this might imply the parent's directory rectangle in turn be enlarged. Thus, a leaf enlargement propagates up the tree, in the worst case up to the root (function ADJUSTPATH).

If the leaf l in which the object has been inserted is full, a *split* occurs. A new leaf l' is created, and the $M + 1$ entries are distributed among l and l'. How to split the leaf (function SPLIT) is the tricky part of the algorithm (explained later).

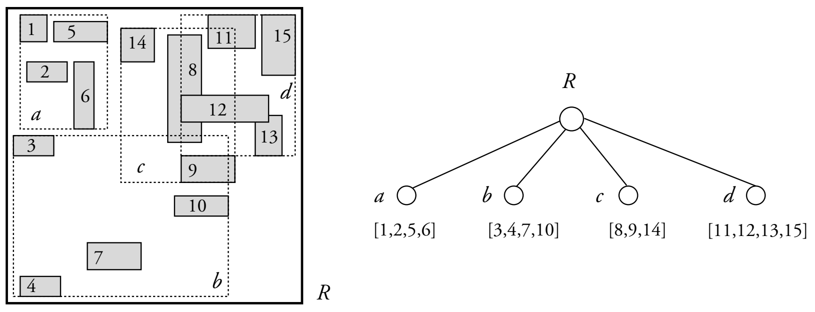


Figure 6.25 Insertion of object 15.

When the split is over, it remains to update the entry of the old leaf l in its parent node f, and to insert a new entry in f corresponding to the new leaf l'. If f itself is full because of this new insertion, the same splitting mechanism is applied to the internal node. In the worst case, this splitting process propagates up the tree to the root (function SplitAndAdjust). The tree depth is incremented by 1 when the root is split.

We illustrate this process by inserting two new objects in the tree of Figure 6.22. For the sake of illustration, we assume that $M = 4$.

Object 15 (Figure 6.25) has first been inserted in leaf *d*. The *dr* of *d* must be enlarged to enclose this new object, and the *d* entry in the root must be adjusted. Object 16 (Figure 6.26) is to be inserted in leaf *b*. Because *b* already contains the four entries {3, 4, 7, 10}, a node overflow occurs (recall that the capacity M is 4), *b* is split. A new leaf, *e*, is then created, and the five entries are distributed among *b* and *e*.

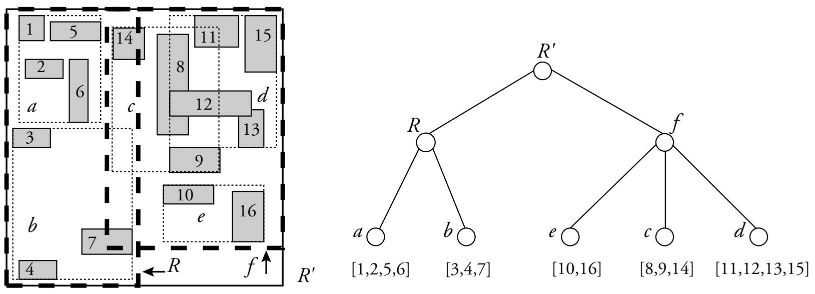


Figure 6.26 Insertion of object 16.

Entry 10 is moved to *e* (where entry 16 had already been inserted). Entries 3, 4, and 7 remain in *b*.

Now a new entry must be inserted in the parent of *b*, *R*. Because *R* already contains four children, it must in turn be split. The process is similar to that described for leaf *b*. A new node *f* is created, *a* and *b* stay in *R*, and *e, c*, and *d* are attached to *f*. Finally, a new root *R′* is created, whose two children are *R* and *f*. The insertion algorithm follows.

ALGORITHM INSERT (*e:* LeafEntry)

```
begin
  // Initializes the search with root
  node = root
  // Choose a path down to a leaf
  while (node is not a leaf) do
    node = ChooseSubtree (node, e)
  end while
  // Insert in the leaf
  InsertInLeaf (node, e)
  // Split and adjust the tree if the leaf overflows, else adjust the path
  if (node overflows) then
    SplitAndAdjust (node)
  else
    AdjustPath (node)
  end if
end
```

Function CHOOSESUBTREE (*node, e*), which is not described here, picks the entry *ne* of *node* such that *ne.dr* either contains *e.mbb* or needs the minimum enlargement to include *e.mbb*. If several entries satisfy this, choose the one with smallest area.

Function ADJUSTPATH propagates the enlargement of the node's *dr* consecutive to an insertion. The process stops when either the *dr* does not require any adjustment or the root has been reached.

ALGORITHM ADJUSTPATH (*node:* Node)

```
begin
  if (node is the root) return
  else
    // Find the parent of node
    parent = GetParent (node)
    // Adjust the entry of node in parent
    if (AdjustEntry (parent, [node.mbb, node.id])) then
      // Entry has been modified: adjust the path for the parent
      AdjustPath (parent)
    end if
  end if
end
```

AdjustEntry (*node, child-entry*) is a Boolean function that looks for the entry *e* corresponding to *child-entry.id* in *node* and compares *e.mbb* with *child-entry.mbb*. If *e.mbb* needs to be updated because the *dr* of the child has been enlarged or shrunken, it is updated, and the function returns **true** (and **false** otherwise).

Function SplitAndAdjust handles the case of node overflow. It performs the split and adjusts the path. It uses functions AdjustPath and AdjustEntry (already described), and can be recursively called in the case of propagated overflows.

ALGORITHM SplitAndAdjust (*node:* Node)

```
begin
  // Create a new node and distribute the entries
  new-node = Split (node)
  if (node is the root) then
    CreateNewRoot (node, new-node)
  else
    // Get the parent of node
    parent = GetParent (node)
    // Adjust the entry of node in its parent
    AdjustEntry (parent, [node.id, node.mbb])
    // Insert the new node in the parent.
    InsertInNode (parent, [new-node.mbb, new-node.id])
    if (parent overflows) then
      SplitAndAdjust(parent)
    else
      AdjustPath (parent)
    end if
    end if
end
```

CREATENEWROOT(*node, new-node*) allocates a new page *p* and inserts into *p* two entries: one for *node* and one for *new-node*. Observe that the root is the only node that does not respect the minimal space utilization fixed by *m*.

Once the entries have been distributed among the two nodes, the entry of *node* must be adjusted in *parent* (call of function ADJUSTENTRY). Then the new node is inserted into the parent node. Two cases may occur. Either the parent node overflows—in which case it must be split and adjusted—or it is only adjusted.

An important part of the algorithm is the *split* of a node. Recall the constraint of a minimal number of entries *m* in a node. Any solution with $m + i$ entries in one node and $M + 1 - m - i$ in the second, with $0 \leq i \leq M - 2m + 1$, is acceptable.

There are many ways of splitting a node. A well-designed splitting strategy should obey the following requirements:

- Minimize the total area of the two nodes.
- Minimize the overlapping of the two nodes.

Unfortunately, these two requirements are not always compatible, as illustrated in Figure 6.27. In the following, we focus on the first criterion.

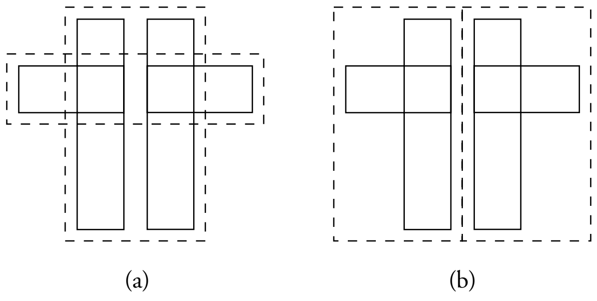


Figure 6.27 Minimal area and minimal overlap: a split with minimal area (a) and a split with minimal overlap (b).

A brute-force method is to exhaustively look at all possible distributions of rectangles among the two nodes and choose the one that minimizes the total area. Because M is of the order of 100 for common page sizes, this is clearly too expensive, the cost of this method being exponential in M.

Therefore, a trade-off between computation time and quality of the result has to be found. We now present two algorithms along these lines. The first, known as a *quadratic split* algorithm, has a computation time quadratic in M. It relies on the following heuristic. Two groups of entries are first initialized by "picking" two "seed" entries e and e', far away from each other, and more precisely such that the dead space is maximal. The dead space is defined as the area of the *mbb* of e and e' minus the sum of the areas of e and e'.

Then the remaining $M - 2$ entries are inserted in one of the two groups as follows. We call *expansion* of a group for an entry the dead space that would result from inserting the entry in this group. Then at each step we look for the entry e for which the difference in group expansions is maximal. We choose to add e to the group "closest" to the entry (for which the expansion is the smallest). In the case of equality, the following secondary criteria can be used: (1) choose the group with the smallest area, and (2) choose the group with the fewest elements.

ALGORITHM QUADRATICSPLIT (E: set of entries)

```
begin
   MBB J; integer worst = 0, d1, d2, expansion = 0
   // Choose the first element of each group,
   // E' is the set of remaining entries
   for each e in E do
      for each e' in E do
         J = mbb(e.mbb, e'.mbb)
         if ((area(J) − area(e.mbb) − area(e'.mbb)) > worst) then
            worst = (area(J) − area(e.mbb) − area(e'.mbb))
            $G_1$ = {e}
            $G_2$ = {e'}
         end if
      end for
   end for
```

```
  // Each group has been initialized. Now insert the remaining entries
  $E' = E - (G_1 \cup G_2)$
  while ($E' \neq \emptyset$) do
    // Compute the cost of inserting each entry in $E'$
    for each e in $E'$ do
      d1 = area (mbb(e.mbb, $G_1$.mbb) − e.mbb)
      d2 = area (mbb(e.mbb, $G_2$.mbb) − e.mbb)
      if (d2 − d1 > expansion) then
        best-group = $G_1$; best-entry = e; expansion = d2 − d1
      end if
      if (d1 − d2 > expansion) then
        best-group = $G_2$; best-entry = e; expansion = d1 − d2
      end if
    end for
    // Insert the best candidate entry in the best group
    best-group = best-group ∪ best-entry
    $E' = E' - \{best\text{-}entry\}$
    // Each group must contain at least m entries.
    // If the group size added to the number of remaining
    // entries is equal to m, just add them to the group.
    if ($|G_1| = m - |E'|$) then $G_1 = G_1 \cup E'$; $E' = \emptyset$
    if ($|G_2| = m - |E'|$) then $G_2 = G_2 \cup E'$; $E' = \emptyset$
  end while
end
```

The two parts of the algorithm (group initialization and insertion of entries) are both quadratic in M. In the last part of the algorithm, if it turns out that one of the groups (say, G_1) has been preferred to the other (say, G_2), the set of remaining entries E' must be assigned to G_2 independently from their location. This might lead to a bad distribution in some cases. Obviously, this depends on the parameter m, which defines the minimal cardinality of a group. It seems that $m = 40\%$ is a suitable value for this algorithm.

There exists a linear algorithm that consists of (1) choosing the seeds such that the distance along one of the axes is the greatest, and (2) assigning each remaining entry to the group whose *mbb* needs the least area enlargement. The algorithm is simpler and faster. However, as reported in several experiments, its result often suffers from an important overlap.

The deletion of an entry e from an R-tree is carried out in three steps: (1) find the leaf L that contains e, (2) remove e from L, and

(3) reorganize the tree if needed. Steps 1 and 2 are similar to the window query previously presented. The reorganization of the tree is the difficult part of the algorithm. It occurs when, after removal of an entry from a node N, N has less than m entries.

A simple approach is to delete the node and to insert the remaining $m - 1$ entries. The deletion algorithm follows.

ALGORITHM DELETE (*e:* LeafEntry)

```
begin
  // Find the leaf containing e
  L = FINDLEAF (e)
  // Remove entries and reorganize the tree, starting from L and e
  // The result is a set Q of nodes
  Q = REORGANIZE (L, e)
  // Reinsert entries in the nodes of Q
  REINSERT (Q)
end
```

The REORGANIZE function removes nodes that contain less than m entries. In the worst case, this can involve all levels of the tree. The function returns the set of removed nodes whose entries must be reinserted.

ALGORITHM REORGANIZE (*N:* node, *e:* entry): set of node entries

```
begin
  Q: set of nodes, initially empty
  // Remove e from N
  $N = N - \{e\}$
  if (N is not the root) then
    if ($|N| < m$) then
      $Q = Q \cup \{N\}$
      // Get the parent and reorganize it
      $F = $ GETPARENT (N)
      $Q = Q \cup$ REORGANIZE (F, entry of N in F)
    else
      ADJUSTPATH (N) // N has been modified: adjust the path
    end if
  end if
```

```
   return $Q$
end
```

Once all nodes to be reinserted have been collected in Q by recursive calls (function REORGANIZE), the elements of Q are reinserted (function REINSERT). Reinserted elements may be either leaf entries or node entries, which must be inserted at the appropriate level in the tree.

The reinsertion process during deletion allows one to obtain a better clustering of rectangles and thus a better spatial organization of the R-tree. However, the overall quality highly depends on the insertion order. An initial creation of the tree with randomly distributed rectangles often results in an important overlapping of nodes. Indeed, once assigned to a node, an entry remains forever in the same part of the tree, although it could possibly be better located later on.

Consider the example in Figure 6.28. In step *a*, four objects are stored in a compact node *u*. Upon insertion of object 5, a split occurs. There is no good choice for the distribution of objects between node *u* and the new node *v*. Any distribution results in a large dead space in node *v*. In Figure 6.28 (step *b*), object 4 has been assigned with object 5 to node *v*. The subsequent insertions (objects 6 and 7, step *c*) do not change the situation, although a better distribution could then be possible.

The following explores two constructions that intend to overcome the poor spatial organization of R-trees due to random insertions. The R*tree relies on entry reinsertion to improve the clustering of *mbb*s, whereas *packed R-trees* pre-process a given set of rectangles to get the best insertion order.

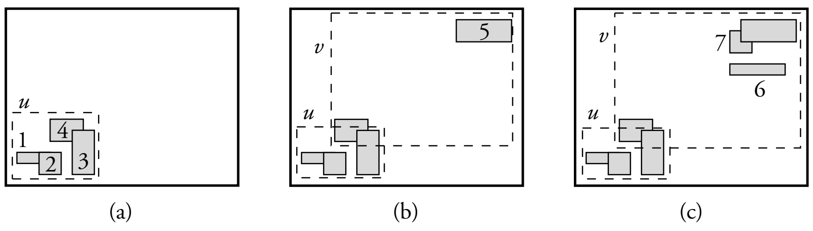


Figure 6.28 A bad insertion order.

6.3.2 *The R*Tree*

The R*tree is a variant of the R-tree that provides several improvements to the insertion algorithm. Essentially, these improvements aim at optimizing the following parameters: (1) node overlapping, (2) area covered by a node, and (3) perimeter of a node's directory rectangle. The latter is representative of the shape of the rectangles because, given a fixed area, the shape that minimizes the rectangle perimeter is the square.

There are no well-founded techniques to simultaneously optimize these three parameters. Thus, the design approach consists of considering each module of the insertion algorithm and experimenting with several heuristics. The following presents two variants that bring the most significant improvement with respect to the original R-tree. The first improves the split algorithm. Recall that when a node of capacity M overflows, a new page must be allocated, and the $M + 1$ entries have to be distributed among the two nodes. Because the size of the solution space (i.e., the number of possible solutions) is exponential in M, one cannot exhaustively look at all solutions for finding the best distribution.

The R-tree split algorithm first initializes the two groups with the two entries that are as far as possible from each other, then assigns each of the remaining entries to a group. The R*tree approach is different in the sense that it assumes the split to be performed along one axis (say, horizontal), and explores all possible distributions of objects above or below the split line.

The expected advantages of this method are illustrated in Figure 6.29 (assuming $M = 4$ and $m = 2$). The R-tree split algorithm chooses A and B as "seeds" for groups G_1 and G_2, respectively. Because A is much

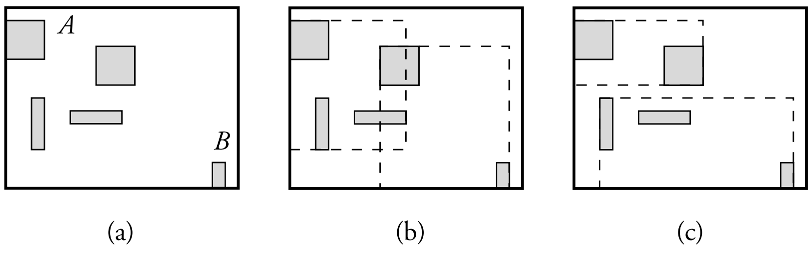


Figure 6.29 Splitting strategies: overflowing node (a), a split of the R-tree (b), and a split of the R*tree (c).

larger than B, subsequent insertions will tend to favor the group of A, G_1. However, when three rectangles have been assigned to G_1, the remaining entries are put in G_2, whatever their position. This results in a clearly non-optimal split.

On the other hand, the R*tree chooses the x axis and finds a partition in two nodes without overlapping. In order to find the best axis, one can sort on the upper or lower value of each rectangle. This yields $2(2(M - 2m + 2))$ computations. Now the best axis is the one such that S, the sum of perimeters, is minimal.

ALGORITHM R*TREECHOOSEAXIS (E: set of entries)

```
begin
  Set variable min-perimeter to the maximum possible value
  for each axis a do
    sort E with respect to a
    $S = 0$
    // Consider all possible distributions
    for $k = 1$ to $(M - 2m + 2)$ do
      $G_1 = E[1, m + k - 1]$; $G_2 = E[m + k, M + 1]$
      // The first $m + k - 1$ entries are stored in $G_1$, the remaining in $G_2$.
      Mg = perimeter (mbb($G_1$)) + perimeter (mbb($G_2$)) // Perimeter value
      $S = S + Mg$ // Compute the sum of the perimeters for the current axis
    end for
    // If S is the minimal value encountered so far, then a becomes the best
      axis
    if ($S <$ min-perimeter) then
      best-axis = a
      min-perimeter = S
    end if
  end for
  return best-axis
end
```

Given the best axis, one chooses the distribution with minimal overlap. If two distributions share the same minimal overlap, one chooses the one with minimal area. These indicators can be computed together with S in R*TREECHOOSEAXIS.

The second important improvement in the R*tree insertion algorithm is the forced reinsertion strategy. Indeed, as seen previously, the

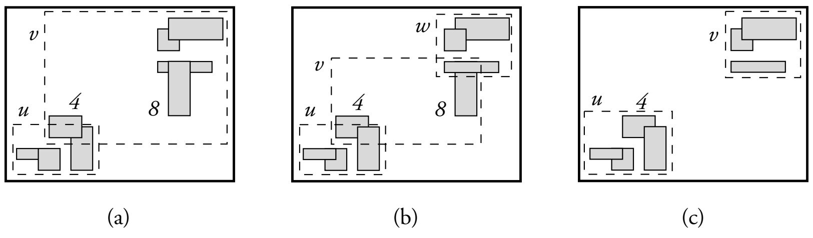


Figure 6.30 The R*tree reinsertion strategy: insertion of 8 (v overflows) (a), a split of the R-tree (b), and R*tree forced reinsertion of 4 (c).

insertion order can dramatically influence the quality of the R-tree organization. The R*tree tries to avoid the degenerated cases (such as those in Figure 6.28) by reinserting some entries whenever a node overflows.

Assume (see Figure 6.30a) that rectangle 8 is inserted in the tree of Figure 6.28c. Node v overflows, and the R-tree split algorithm will only perform a local reorganization with a rather important node overlapping (Figure 6.30b). The R*tree tries to avoid splitting by reinserting the rectangles in v for which the dead space in the node is the largest; here, rectangle 4. The reinsertion algorithm proceeds as follows:

- Remove 4 from v.
- Compute the new bounding box of v.
- Reinsert 4, starting from the root: the CHOOSESUBTREE algorithm assigns 4 to node u.
- Now entry 8 can be put into v, and no splits occur.

Because a node overflow might occur at any level of the tree, a removed entry must be reinserted at the same level it was initially inserted. This was already required by the DELETE function of the R-tree. However, we must avoid entering an infinite loop by reinserting the entries in their initial node. Hence, if we detect that a same node overflows a second time during the insertion of a given rectangle, the split algorithm must be carried out.

The entries to be reinserted are chosen as follows: compute the distance d between their centroid and the centroid of the node's *mbb*, sort in decreasing order on d, and take the p first entries from the sorted list. It seems that the best performance is obtained when $p = 30\%$ of the entries of an overflowing node are reinserted.

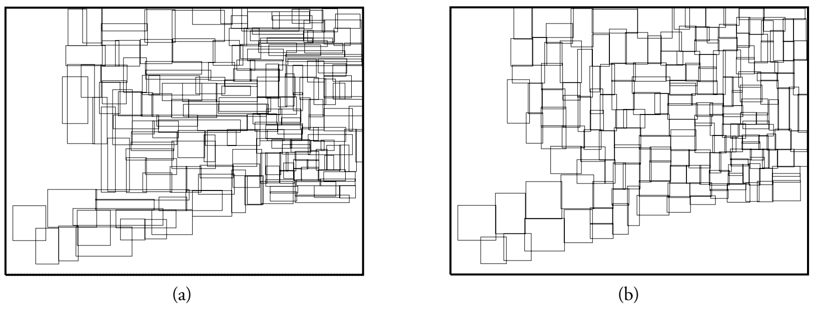


Figure 6.31 Comparison of R-tree (a) and R*tree (b) (state of Connecticut).

The improved split algorithm and the reinsertion strategy result in a much better organization of the R*tree with respect to the original R-tree, as illustrated in Figure 6.31, which gives both for the R-tree and the R*tree the *dr* of the leaves for the same collection of rectangles.

It is worth remembering that the data structures for the R-tree and R*tree are the same. Hence, the data retrieval operations defined for the R-tree remain valid with the R*tree. Due to the better organization of the R*tree, the performance of these operations is likely to be significantly improved.

6.3.3 *R-Tree Packing*

The algorithms presented so far for inserting entries in an R-tree (or an R*tree) are *dynamic* in the sense that they enable insertions and deletions in an already existing R-tree. However, the evolution of the structure with time does not allow one to optimize the space utilization and thus might lead to a performance degradation as the number of insertions and deletions grows.

In the *static* case, when the collection of rectangles is stable with time, one can pre-process the data before creating the R-tree. Several algorithms, called *packing algorithms*, have been proposed for the R-tree. The idea behind packing is best understood by looking at the B-tree case. If the data set to be indexed is known a priori, we can *sort* data, and construct the index bottom-up. First, the leaves are sequentially filled. The first M entries go to the first leaf, the M following to the second leaf, and so on. Once the leaf level has been created, the operation is recursively repeated on the following level up to the root. One obtains a B-tree with nearly 100% space utilization.

In the case of the R-tree, the pre-processing phase must sort the rectangles according to their location. We describe in the following a simple and efficient algorithm, called the Sort-Tile-Recusive (STR) algorithm.

The STR algorithm is as follows. Assume that the size of the data set is N. Then the minimal number of leaves is $P = \lceil N/M \rceil$, where M denotes, as usual, the node capacity. The idea is to organize the leaves as a "checkerboard" having vertical and horizontal slices. In addition, we require that the number of vertical slices be the same as the number of horizontal slices and equal to $\lceil \sqrt{P} \rceil$.

We first sort the source rectangles on the x coordinate of their centroid, and we partition the sorted list into $\lceil \sqrt{P} \rceil$ groups. This defines the vertical slices. Then the rectangles of each slice are loaded, sorted by y coordinate of their centroid, and grouped into runs of size M. This defines the leaves of the packed R-tree. Finally, we take the set of directory rectangles of the leaves and construct recursively the upper levels according to the same algorithm. The organization in horizontal or vertical slices is visualized in Figure 6.32, on the Connecticut hydrography data set.

All leaves (except those of the last vertical slice) are completely filled, so as to maximize space utilization. This may be a drawback if we want to allow subsequent dynamic insertions. Indeed, a split will occur at each level of the tree as soon as an object is inserted. Therefore, a space utilization lower than 100% should be chosen in an actual implementation.

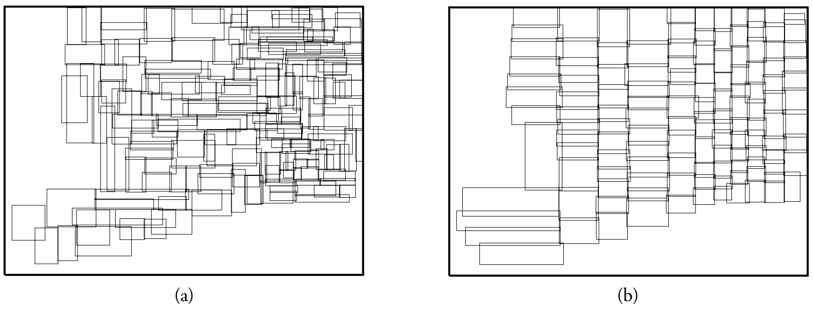


Figure 6.32 R-tree (a) and STR packed R-tree (b).

In spite of the optimized construction, there is still some (unavoidable) overlapping among the nodes. We conclude the presentation of the R-tree with the R+tree variant, which generates no node overlapping.

6.3.4 *The R+Tree*

In the R+tree, the directory rectangles at a given level do not overlap. This has as a consequence that for a point query a single path is followed from the root to a leaf. Unlike the previous variants, the I/O complexity of this operation is thus bounded by the depth of the tree. The R+tree is defined as follows:

- The root has at least two entries, except when it is a leaf.
- The *dr* of two nodes at the same level cannot overlap.
- If node *N* is not a leaf, its *dr* contains all rectangles in the subtree rooted at *N*.
- A rectangle of the collection to be indexed is assigned to all leaf nodes the *dr*s of which it overlaps. A rectangle assigned to a leaf node *N* is either overlapping *N.dr* or is fully contained in *N.dr*. This duplication of objects into several neighbor leaves is similar to what we encountered earlier in other space-driven structures.

Figure 6.33 depicts an R+tree. Note that objects 8 and 12 are referenced twice. Object 8 is overlapping leaves *p* and *r*, whereas object 12 is overlapping leaves *p* and *q*. Note also that both at the leaf level and at the intermediate level, node *dr*s are not overlapping.

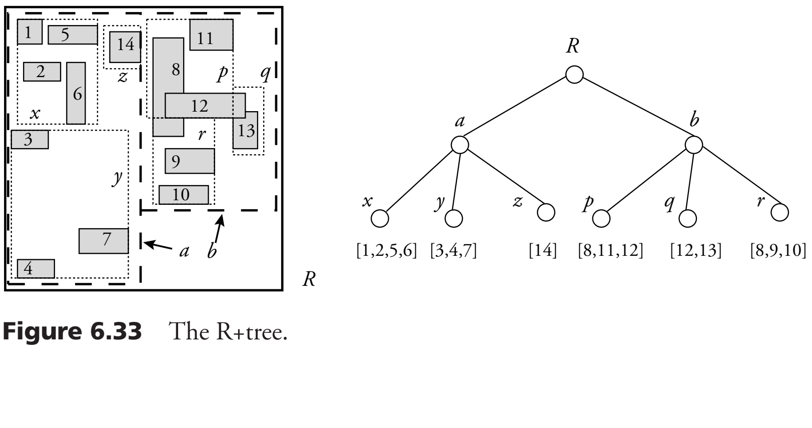


Figure 6.33 The R+tree.

The structure of an R+tree node is the same as that of the R-tree. However, because we cannot guarantee a minimal storage utilization m (as for the R-tree), and because rectangles are duplicated, an R+tree can be significantly larger that the R-tree built on the same data set.

The construction and maintenance of the R+tree are rather more complex than with the other variants of the R-tree. The following briefly outlines the dynamic insertion of an object.

At each level of the tree, a rectangle is inserted in all nodes the *dr*s of which it overlaps. In order to achieve a zero overlap among the directory rectangles at the same level, the R+tree uses *clipping*. When a rectangle R to be inserted overlaps several directory rectangles $\{r_1, \ldots r_n\}$, R is split into subrectangles, one per r_i it overlaps, and inserted recursively in each of the paths starting at these r_is. Therefore, the same object will be stored in several leaves of the tree (duplication). For instance, in Figure 6.34a, R must be inserted in the subtrees rooted at r_1 and r_2.

If R is not fully contained in the union of the r_is it overlaps, one or several of the directory rectangles must be enlarged. This is similar to the R-tree insertion algorithm (see the AdjustPath function), except that we must now prevent the enlarged nodes from overlapping. Unfortunately, this is not always possible (illustrated in Figure 6.34b), as none of the directory rectangles r_1, r_2, r_3, and r_4 can be enlarged without introducing some overlap. In that case, some reinsertion must be carried out.

Finally, when a node overflows, it has to be split. Once again, this is similar to the R-tree, except that the split must be propagated downward (see Figure 6.34c), still because of the nonoverlapping rule. This complicates the algorithms, and deteriorates the space utilization. For instance, in Figure 6.34c, the node A is split. Then the two children b and c must be split as well.

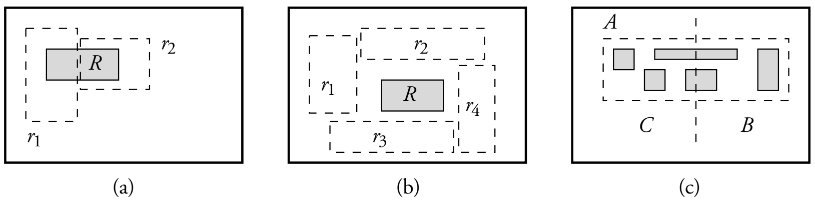


Figure 6.34 Operations on the R+tree: clipping in subrectangles (a), a deadlock (b), and a downward split propagation (c).

In summary, the point query performance benefits from the nonoverlapping of nodes. As for space-driven structures, a single path down the tree is followed, and fewer nodes are visited than with the R-tree. The gain for window queries is less obviously assessed. Object duplication not only increases the tree size, but potentially leads to expensive post-processing of the result (sorting for duplication removal).

6.3.5 *Cost Models*

We end the R-tree description with the presentation of a cost model. *Cost models* are useful for predicting the performance of an operation as a function of physical parameters of the spatial database, such as size of indices, size of records, selectivity of selection predicates, and so on. The following is a simple model for estimating the performance of operations on R-trees.

In order to simplify the presentation, we assume (without loss of generality) that the search space is a normalized unit square $[0, 1]^2$. Now consider a node N_i of an R-tree R, with a directory rectangle $N_i.dr$ of size $(s_{i,x}, s_{i,y})$, and assume that we execute point queries on the R-tree with uniformly distributed point arguments. Then the probability for a point query to fall into node N_i is

$$PQ(N_i) = s_{i,x} \times s_{i,y}$$

In other words, the probability is the ratio of the size of $N_i.dr$ and the size of the work space (which is equal to 1). Now the R-tree nodes visited during a point query are those whose dr contains the point argument. The number of visited nodes can be expressed as

$$\begin{aligned} PQ(R) &= \sum_{i=1}^{n} PQ(N_i) \\ &= \sum_{i=1}^{n} s_{i,x} \times s_{i,y}, \end{aligned}$$

where n denotes the number of nodes in R. These formulas can be extended to estimate the cost of window queries. Assume we are given a node N and a window w with size (w_x, w_y). The idea is to estimate the probability that $N.mbb$ intersects w as the probability of a point

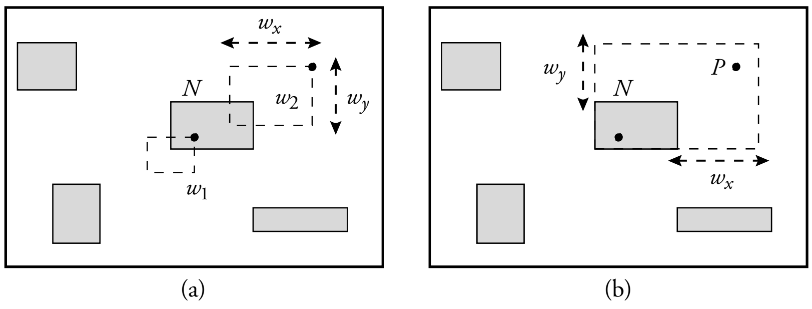


Figure 6.35 Estimating the cost of window queries: a window query (a) and its equivalent point query (b).

query with the upper-right corner of w. Indeed, two cases may happen (see Figure 6.35a):

- The upper-right corner of the window falls into $N.mbb$ (window w_1)
- The upper-right corner of w falls into an "inflation" of $N.mbb$, with w_x on the right and w_y above (window w_2)

It follows that a window w intersects $N.mbb$ if and only if its upper-right corner falls into the rectangle $W = (N.mbb.x_{min} + w_x, N.mbb.y_{min} + w_y)$. Because we assume that the window arguments are uniformly located in the search space, the cost can be estimated as for a point query over W (Figure 6.35b). In summary, the number $WQ(w)$ of nodes retrieved during a window query is

$$\begin{aligned} WQ(w) &= \sum_{i=1}^{n}(s_{i,x} + w_x) \times (s_{i,y} + w_y) \\ &= \sum_{i=1}^{n} s_{i,x} \times s_{i,y} + w_y \times \sum_{i=1}^{n} s_{i,x} \\ &\quad + w_x \times \sum_{i=1}^{n} s_{i,y} + n \times w_x * w_y \end{aligned}$$

This formula depends only on the properties of the R-tree, whatever its specific type (R-tree, R*tree) or construction mode (static or dynamic).

6.4 BIBLIOGRAPHIC NOTES

The design of 2D spatial structures for main memory was a flourishing research field in the computational geometry community in the 1970s. The need for external memory data structures, motivated by applications such as geographic applications, gave rise to an important research activity, both in the database and in the GIS communities (which started in the late 1970s). The classification of SAM is difficult because of the large number of parameters involved. The space-driven versus data-driven classification of Section 6.1 is common, but too simple to take into account the large variety of structures proposed.

There exist a large number of comprehensive studies on SAM and related issues, among which [Sam90b, Sam90a] is one of the first textbooks. It focuses, however, on space-driven access methods. Other general references of interest are [NW97, Aga97, GMUW00, MTT99]. A complete and recent survey of multidimensional access methods can be found in [GG98]. The latter reference distinguishes between point access methods (PAMs) and access methods for rectangles (SAMs). It also discusses transformation techniques that map a collection of rectangles onto a collection of 4D points that can then be managed by a point access method.

The B-tree proposed by Bayer and McCreight was first described in [BM72, Com82]. It has been utilized in numerous system designs, including all commercial relational DBMSs, because of its optimal performance. Lately, some extensions of the B-tree have been proposed for multidimensional access methods. The BV-tree [Fre95] is an attempt to generalize the B-tree to higher dimensions. A weight-balanced B-tree is proposed in [AV96]. The Cross-tree [GI99] is a multidimensional version of a B-tree that combines the B-tree data-driven partition of data with the quadtree space-driven partitioning.

PAMs index points and therefore do not need a decomposition in two steps: filter step and refinement step. In contrast, SAMs deal with lines and regions, approximated by rectangular *mbb*, and there is a clear division between the filter step that deals with *mbb* and the refinement step on the exact geometry. The approximation of lines by *mbb* is not appropriate in many applications. Although the *mbb* approximation of objects has been widely accepted, there have been a number of other proposals for region indexing (see [GG98]). In the polyhedral trees of

[Jag90b], for example, objects are approximated by a polygon rather than just a rectangle. Region objects have no polygonal approximation in the cell tree proposed by Günther [Gün89]. Instead, the search space is decomposed into disjoint convex subspaces. Convex components of the region to be inserted in the index are stored in the tree leaves.

The simplest space-driven SAM is the fixed grid, proposed by Bentley and Friedman [BF79]. The grid file was proposed for point indexing by Nievergelt, Hinterger, and Sevcik [NHS84]. There are a large number of extensions of the grid-based SAM (see [GG98]). Among them it is worth mentioning the multilayer grid file [SW88] and the BANG file [Fre87]. The latter is a PAM that partitions the search space in rectangular regions like the grid file, but allows the regions to overlap.

Finkel and Bentley [FB74] proposed one of the first variants of the quadtree for points. Another variant for points, called a region quadtree, is proposed by Samet [Sam90b]. There exist a large number of quadtrees for storing polygons and rectangles, described in [Sam90b].

A number of authors have proposed to map a simple space-driven index, most of the time a quadtree, onto a B+tree [Gar82, AS83, OM84, GR94, AS95, SGR96, FR89, Jag90b]. The linear order on cells in most proposals is based on *Peano curves* or *Morton code*, also referred to as *z-ordering* [OM84]. Some authors propose the use of different space-filling curves (not only for space-driven structures, see [GG98]), such as the Hilbert curve [FR89, Jag90b] or Gray codes [Fal86, Fal88]. The linear structures vary according to the assignment of objects to cells. The first scheme of Section 6.2.2, in which an object is assigned to all cells overlapping its *mbb*, is from [SGR96], where the linearized SAM is either a quadtree or a grid. The original SAM called *z*-ordering due to Orenstein and Merret, in which an object is approximated by a union of cells, was first described in [OM84] and further studied in [Ore86, OM88, Ore89, Ore90]. The redundancy of such a structure is analyzed in [Ore89, Gae95]. For clarity, our presentation of *z*-ordering uses a canonical quadrant numbering from 0 to 3. However, this coding allows us to gather only four sibling subquadrants into a single larger element, whereas the original *z*-ordering code for quadrants (based on bit interleaving) allows a finer grouping of two neighbor horizontal or vertical quadrants into a larger element. *z*-ordering is also used in [GR94].

The HHcode of the Oracle Spatial Data Option (SDO) [Sha99] is also a variant of *z*-ordering mapped onto the Oracle kernel B+tree. Two variants are proposed by Oracle SDO (see Chapter 8), among which one corresponds to the raster variant of *z*-ordering (see Section 6.2.3). The third scheme presented in Section 6.2.3, in which an object is assigned to the smallest quadtree quadrant containing it, is from [AS83]. As a matter of fact, in [AS83], *mbb*s rather than objects are indexed. This enables one to eliminate false hits prior to the refinement step.

The loss in clustering due to the mapping of cells onto a linear structure is important in range queries (see window query of the linear quadtree, Section 6.2.2). There have been several proposals for better performing, although more complex, window queries [AS95, SGR96]. [AS95] is based on *maximal quadtree blocks* [AS93]. [SGR96] maps a fixed grid and a quadtree (rectangles are assigned to each overlapping cell or quadrant) with optimized window querying onto a B+tree and compares their performance. Finally, [BKK99] proposes a scheme called XZ-ordering, a variant of *z*-ordering that allows a large resolution and thus a better approximation without performance degradation. Object duplication is avoided by enabling the overlap of cells, due to a new coding scheme for cells. Several improvements to the *z*-ordering technique are also surveyed in [BKK99].

The R-tree was introduced by Guttman [Gut84]. In this chapter, we closely followed the author's presentation for the insertion algorithm.

The R*tree variant was designed by Beckman, Kriegel, Schneider, and Seeger [BKSS90]. Section 6.3 illustrates the impact of the splitting algorithm on the R-tree performance. In [BKSS90], a performance comparison with several earlier versions of the R-tree (including the quadratic and linear R-tree) is made for several types of queries on a large range of data sets. The R*tree is shown to be steadily the best. It should be noted, however, that the R*tree construction is sometimes reported (for instance, in [HS92]) as being more costly than the R-tree construction with quadratic split, although the experiments in [BKSS90] did not show a significant overhead in I/Os.

The issue of R-tree node splitting has been revisited in [GLL98]. The paper proposes an optimal polynomial algorithm [in $O(n^d)$, where d is the dimension of the input] and a new dynamic insertion method.

Packing R-trees was successively proposed in [RL85], [KF94], and [LEL96]. The latter paper describes and experiments with the STR variant presented in Section 6.3. A new algorithm for fast bulk loading of the R*tree is described in [dBSW97]. Sellis, Roussopoulos, and Faloutsos proposed the R+tree variant [SRF87].

R-trees were primarily designed for supporting point and range (window) queries. Some authors consider other operations. [RKV95] studies the problem of reporting the nearest neighbors of a point P using R-trees. Given the *mbb* R of an object O, two distance functions of (R, P) are defined, which give, respectively, a lower and an upper bound of the distance from P to O. These functions are used during the traversal of the R-tree to explore the search space. An analytical study of the nearest neighbor problem is reported in [PM97]. Topological queries are considered in [PTSE95]. Topological information conveyed by the *mbb* of an R-tree is analyzed and used to optimize the processing of queries that involve one of the topological predicates of the 9-intersection model (see Chapter 3).

An analytical estimate of the search performance with R-trees was first proposed in [FSR87]. The formulas given in Section 6.3.5 are from [KF93], and were independently presented in [PSTW93]. They rely on the strong assumption that data is uniformly distributed. Subsequent works relaxed this assumption. See, for instance, [FK94] and [TS96]. Extensions of cost models to spatial joins can be found in [HJR97a, TSS98, PMT99].

Another class of tree structures for SAMs originated as modifications of the kd-tree [Ben75], suitable for secondary storage, such as the k-d-b-tree [Rob81] and the hB-tree [LS90].

Lately, external memory algorithms have again received considerable attention from the computational geometry community (see [Vit98] for a recent survey on external memory algorithms). This interest has two motivations. First, new nonstandard applications require efficient SAMs (as, for instance, spatio-temporal applications), in particular those dealing with *moving objects*, or network applications dealing with polyline objects and constraint databases requiring efficient access to constraint-based objects. The second motivation is to approach the optimal search time complexity. Consider first the latter objective. Getting, as in the one-dimensional case with B-trees, an optimal multidimensional access method is a real challenge. Recall an optimal SAM is such that its search complexity is $O(log_B(N) + T/B)$, where N is the collection size, B the buffer size, and T the result size; its update complexity is $O(log_B(N))$; and its space complexity is $O(N/B)$.

All SAMs proposed until now and discussed in this chapter are not optimal in worst-case conditions. Because the problem is not simple, most current theoretical studies provide close to optimal or optimal solutions for simpler problems, and queries such as the *stabbing query* (find all intervals of a line containing a point) [AV96]. The latter is a key component in the *dynamic interval management problem* (find all intervals that intersect a query interval on a given line; see [AV96, Vit98]). Because a large number of efficient main memory data structures have been designed for this problem and related issues, the trend has been in the past decade to adapt these techniques to multidimensional access methods. The range tree [Ben80], the segment tree [Ben77], the interval tree [Ede83], and the priority searchtree [McC85] are elegant main memory structures for 2D range searching. The previously cited theoretical work is justified by the three fields of application: moving objects, network applications, and constraint databases.

Network applications are common applications dealing with polyline objects. Approximating a polyline, even if it has been cut into smaller pieces, by a rectangle is not efficient, because most of the rectangle space is lost. Some classical SAMs have been adapted to line indexing [Sam90b]. Examples of SAMs dedicated to line segments are [Jag90a, KS91, BG94]. In the moving object application [SWCD97, WXCJ98, KGT99], a point is following a fixed trajectory (e.g., belonging to a road network) or a free line trajectory in the plane. A typical query is “Report the objects inside a rectangle within a given time interval.”

Moving object applications are just one example of spatio-temporal applications that may require specific indexing techniques [TSPM98]. A recent work proposes an adaptation of the R-tree to moving object indexing [SJLL00]. The basic idea is that moving objects must be indexed by moving rectangles that at each moment enclose their entries. The parameter to minimize is then the sum of the areas (or, to better say, the integral) of the moving rectangles during their motion. The paper revisits the split algorithms of the R*tree in this context. There is a growing interest in the problem of moving point indexing. See, for instance, [NST99, KGT99, PJT00] and [TUW98], which attempts to use quadtrees. In order to support this research, some authors provide spatio-temporal data set generators [SM99, TSN99], following a trend initiated by [GOP$^+$98] with the *À la Carte* generator. This tool generates a set of rectangles according to a statistical model whose parameters (size, coverage, and distribution) can be specified.

Last, but not least, constraint databases motivated most of the theoretical work currently undertaken toward efficient SAMs for solving the interval management problem. Kanellakis, Ramaswamy, Vengroff, and Vitter [KRVV96] are at the origin of the work. They relate the problem of constraint indexing to the dynamic interval management problem and provide for this an index called the metablock tree. This work has initiated a large number of ongoing studies. [Ram00] is a comprehensive paper on the topic. See also [Vit98] for a survey on related external memory algorithms.