CAR-TR-989 August 2003
CS-TR-4523
UMIACS 2003-91

Decoupling Partitioning and Grouping: Overcoming Shortcomings of Spatial Indexing with Bucketing

Hanan Samet

Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

Abstract

The principle of decoupling the partitioning and grouping processes that form the basis of most spatial indexing methods that use tree directories of buckets is explored. The decoupling is designed to overcome the following drawbacks of traditional solutions:

1. Multiple postings in disjoint space decomposition methods that lead to balanced trees such as the hB-tree where a node split in the event of node overflow may be such that one of the children of the node that was split becomes a child of both of the nodes resulting from the split.
2. Multiple coverage and nondisjointness of methods based on object hierarchies such as the R-tree which lead to nonunique search paths.
3. Directory nodes with similarly-shaped hyper-rectangle bounding boxes with minimum occupancy in disjoint space decomposition methods such as those based on quadtrees and k-d trees that make use of regular decomposition.

The first two drawbacks are shown to be overcome by the BV-tree where as a result of decoupling the partitioning and grouping processes, the union of the regions associated with the nodes at a given level of the directory does not necessarily contain all of the data points although all searches take the same amount of time. The BV-tree is not plagued by the third drawback. The third drawback is shown to be overcome by the PK-tree where the grouping process is based on ensuring that every node has at least *k* objects or blocks. The PK-tree is not plagued by the first two drawbacks as they are inapplicable to it. In both cases, the downside of decoupling the partitioning and grouping

The support of the National Science Foundation under Grants EIA-99-00268, IIS-00-86162, and EIA-00-91474, and Microsoft Research is gratefully acknowledged.

processes is that the resulting structure is not necessarily balanced, although, since the nodes have a relatively large fanout, the deviation from a balanced structure is relatively small.

Keywords: Spatial indexing, decoupling, space decomposition, object hierarchies, BV-trees, PK-trees, R-trees.

1 Introduction

The ability to handle data with a locational component efficiently is becoming a crucial requirement for conventional databases. It is no longer just a requirement in geographic information systems (GIS) which are part of the more general area of spatial databases. In particular, many organizations increasingly find that virtually all of their data has some spatial aspect. For example, it has recently been mandated (E911 law) that all cell phones be able to provide the location of the user in the case of calls for emergency help [1, 2]. This means that spatial data must be capable of being accessed quickly. Such data is usually large in volume and thus will naturally reside primarily in secondary storage (e.g., disk) rather than in main memory. In order to be able to perform dynamic updates, the data is usually accessed using some variant of a tree structure. This means that tree access structures for d-dimensional data such as quadtree and k-d tree variants (e.g., [73, 74]), which are based on making between 1 and d tests at each level of the tree, become impractical due to the limited fanout at each node. The problem is that each time we have to descend a level in the tree by following a pointer, we may have to make a disk access. This is far costlier than indirect addressing which is how the levels are descended when the tree resides entirely in main memory. This has led to the development of what are termed *bucket methods*.

Bucket methods have been extensively studied in the database literature especially in the context of the representation of multidimensional points (e.g., [20, 37, 74]). There are many different bucket methods as well as ways to distinguish between them. One way to do so is by noting whether they organize the data to be stored or the embedding space from which the data is drawn. An alternative, and the one we follow in this paper, is to subdivide them into two principal classes. The first class consists of aggregating the data objects in the underlying space. The second class consists of decompositions of the underlying space with an appropriate access structure. It is important to note that not all of the methods can be classified so distinctly in the sense that some of them can be made to fit into both classes. We do not dwell on this issue further here.

Depending on whether the bucket methods organize the data to be stored or the embedding space from which the data is drawn, the implementation of the bucket methods using a tree access structure is characterized by the initial application of a partitioning process to either the collection of objects or the underlying space, respectively, into smaller units followed by the application of a grouping process to form the buckets which are organized by the tree. In most cases the grouping process is the same as the partitioning process (e.g., a region quadtree [47, 74] where the underlying space is recursively decomposed into four congruent homogeneous regions and each nonleaf node in the resulting tree access structure that represents the grouping of the resulting regions has four sons). In this paper we show how some of the classic drawbacks of traditional bucket methods can be avoided by decoupling the partitioning and grouping processes so that they are not necessarily the same. For the bucket methods that are based on a decomposition of the underlying space, this is achieved by the PK-tree [79, 80, 83], while for the bucket methods that are based on an object hierarchy as well as a decomposition of the objects into nonintersecting subobjects, this is achieved by the BV-tree [34].

Note that the term “decoupling” could also be used to describe the differentiation of the construction of a spatial index with the aid of bulk loading (e.g., [10, 18, 51]) from the process of building it incrementally. In other words, the term “decoupling” could be applied to many other processes used in the construction of spatial indexes besides the partitioning and grouping processes. Therefore, whenever we use the term in this paper we make sure to qualify it appropriately unless there is no potential for confusion.

The rest of this paper is organized as follows. Section 2 reviews traditional bucket methods and their drawbacks which motivates the need for decoupling the partitioning and grouping processes. Section 3 describes the PK-tree, while Section 4 describes the BV-tree. Section 5 reviews what we have achieved by decoupling the partitioning and grouping processes as well as contains directions for future research. In addition, there is a brief comparison of the PK-tree and BV-tree with some other related representations. Note that we do not perform an experimental comparison of the PK-tree and the BV-tree as they differ fundamentally, and each achieves its goal of overcoming the drawbacks of its corresponding bucketing method family.

2 Review of Bucketing Methods and Motivation

The first class of bucket methods aggregates the actual data objects into sets (termed *buckets*) usually the size of a disk page in secondary memory so that the buckets are as full as possible. These methods still make use of a tree access structure where each node in the tree is the size of a disk page and all leaf nodes are at the same level. Thus the fanout is much larger than in the quadtree (and, of course, the k-d tree). It should be clear that decomposing the underlying space is not a goal of these representations. Instead, they are more accurately classified as object hierarchies since they try to aggregate as many objects (points in our case) as possible into each node.

The R-tree [39] is the most common representation that is based on an object hierarchy. In order to facilitate the execution of search queries, some variant of a bounding box is associated with each object. The bounding box is usually of a simple geometric shape thereby enabling faster point-in-object or object intersection tests. The most common bounding box is an axis-aligned hyper-rectangle as in the R-tree although other shapes are also used such as hyper-rectangles of arbitrary orientation [22, 38, 68]; minimum bounding spheres as in the SS-tree [82], sphere tree [43, 63], and the balltree [62]; the intersection of a minimum bounding box with a minimum bounding sphere as in the SR-tree [46]; a convex hull [22]; as well as a polygon of a bounded number of sides and orientations [45].

When the nodes are too full, they are split at which time some of the representations (e.g., the R-tree [39]) attempt to aggregate spatially proximate objects in the nodes resulting from the split. However, there is no guarantee that the space spanned by the collections of objects in the nodes is disjoint. In the case of high dimensional data, when there is too much overlap among the nodes corresponding to the partitions that result from a node split, then some variants of the R-tree such as the X-tree [17] do not split the node. Nondisjointness is a problem for search algorithms on object hierarchies as the path from the root of such structures (e.g., an R-tree) to a node that spans the space containing a given data point or object is not necessarily unique. In other words, the data point or object can be covered by several nodes of the R-tree, yet it is found in only one of them. This is known as the *multiple coverage problem* and results in more complex searches. No such problems exist when the decomposition of the space spanned by the collections of objects in the nodes is disjoint. Examples of such methods include the k-d-B-tree [69], the R$^+$-tree [30, 76, 77] (which is really a k-d-B-tree [69] with bounding boxes around the portions of space resulting from the decomposition), the LSD-tree [40], the hB-tree [54, 55, 71] and the BANG file [32]. When the data objects have extent (e.g., nonpoint objects such as rectangles, lines, etc.), the disjointness requirement of these representations may result in decomposing the individual objects that make up the collection at the lowest level into several pieces. The disjointness requirement is usually expressed in the form of a stipulation that the minimum bounding box of the collection of objects at the lowest level of the tree (i.e., at the leaf nodes) are disjoint instead of that the minimum bounding boxes of the individual objects are disjoint[1].

The price that we pay for disjointness is that there is no longer a guarantee that each node will be as full as possible. On the other hand, the disjoint solutions do not suffer from the multiple coverage problem. However, the drawback of the disjoint solutions is that if the data objects have spatial extent (e.g., line segments, rectangles, etc.), then an object *o* may need to be decomposed into several pieces and hence reported as satisfying the query several times as the area spanned by *o* may be contained in several blocks. For example, suppose that we want to retrieve all the objects that overlap a particular region (i.e., a window query) rather than a point. In this case, we could report the same object as many times as it has been decomposed into blocks. We can avoid reporting the object several times when using these methods by removing the duplicate objects before reporting the final answer. Removing the duplicate objects usually requires invocation of some variant of a sorting algorithm. Interestingly, there has been some work in developing algorithms for certain classes of objects and different data structures which are based on a disjoint decomposition that avoid reporting duplicate objects (e.g., [7, 9, 25]) without resorting to sorting. In order to simplify matters, in this paper we only deal with point objects although much of the discussion is also applicable to non-point objects.

The above methods that are based on a disjoint decomposition of the embedding space from which the objects are drawn can also be viewed as elements of the second class of bucket methods. However, as we see below, this class is more general. The tree access structures that are used differ from quadtree and k-d tree access structures in terms of the fanout of the nodes. As in the case of an R-tree, the data points are aggregated into sets (termed *point buckets* for the moment) where the sets correspond to subtrees S of the original access structure T (assuming variants of T with bucket size 1 and where all data points are stored in the leaf nodes). They are made to look like the R-tree by also aggregating the nonleaf nodes of T into buckets (termed *region buckets* for the moment) thereby also forming a multiway tree (i.e., like a B-tree). This is in contrast to the more customary way of applying bucketing to quadtree and k-d tree access structures (e.g., the bucket PR quadtree, bucket PR k-d tree, bucket generalized k-d trie, bucket PMR quadtree, etc. [74]) where only the contents of the leaf nodes are aggregated into buckets — that is, more precisely, the decomposition of the underlying space ceases whenever a region contains at most b points, where b is known as the bucket capacity. Thus in the bucket methods that we consider in this paper, the elements of the region buckets are regions. The tree access structures differ from object hierarchies such as the R-tree in the following respects:

1. the aggregation of the underlying space spanned by the nodes is often implicit to the structure, and
2. all nodes at a given level are disjoint and together they usually span the entire space.

In contrast, for the R-tree, the following must hold:

1. the spatial aggregation must be represented explicitly by storing the minimum bounding boxes that correspond to the space spanned by the underlying nodes that comprise the subtree, and
2. the bounding boxes may overlap (i.e., they are not necessarily disjoint).

A region bucket R, whose contents are nonleaf nodes of T, corresponds to a subtree of T and its fanout value corresponds to the number of leaves in the subtree represented by R. Again, note that

[1] Requiring the minimum bounding boxes of the individual objects to be disjoint may be impossible to satisfy as is the case, for example, for a collection of line segments all of which meet at a particular point.

the leaves in the subtrees represented by the region bucket are nonleaf nodes of the access structure T.

The use of a directory in the form of a tree to access the buckets was first proposed by Knott [48]. In database applications, the most commonly used tree directory is the B$^+$-tree [12, 23], in which case the leaf nodes serve as buckets, and data objects are only stored in the leaf nodes. One simple way to make use of the B$^+$-tree in the context of our application is to employ a decomposition rule such as the bucket PR quadtree [64, 74]. In this case, the underlying space is recursively decomposed into 4 congruent (i.e., square) blocks until each block is either empty or contains at most b points, where b is the bucket capacity. The leaf blocks function as buckets. For example, Figure 1a is the block decomposition induced by a PR quadtree with a bucket capacity of 1 (i.e., $b = 1$) for a set of 8 points, and Figure 1b is its tree representation. In order to store and access the blocks in a B$^+$-tree, we must order them. Such an ordering is obtained by assigning each block a unique number known as a *locational code* (e.g., [73]). For example, such a number can be formed by applying bit interleaving to the binary representations of the coordinate values of an easily identifiable point in each block (e.g., its lower-left corner) and concatenating this number with the base 2 logarithm of its block size — that is, i for a block of size 2^i. Sorting the locational codes in increasing order yields an ordering (called a Morton order [56] or Z-order [65]) which is equivalent to that which would be obtained by traversing the nonempty leaf nodes (i.e., blocks) of the tree representation of the decomposition of the embedding space in the order SW, SE, NW, NE for a quadtree (or bottom, top, left, right for a k-d tree). Of course, other orderings such as the Peano-Hilbert order [41] can also be used (e.g., [29, 44]).

Note that due to the use of regular decomposition (i.e., congruent blocks resulting from the recursive halving process), we can reconstruct the PR quadtree just from knowledge of the size and path from the root of the nonempty leaf blocks (i.e., the locational code of the block), and thus we only need to keep track of these blocks — that is, we ignore the empty and nonleaf blocks. In fact, this is how the B$^+$-tree is usually implemented (e.g., [3, 70]). For example, Figure 1c is one possible B$^+$-tree corresponding to the leaf blocks of the PR quadtree whose block decomposition is given in Figure 1a. Each node of the B$^+$-tree in our example has a minimum of 2 and a maximum of 3 entries. We do not show the values of the locational codes of the contents of the leaf nodes of the B$^+$-tree. We also do not show the discriminator values that are stored in the nonleaf nodes of the B$^+$-tree. We have marked the leaf blocks of the PR quadtree in Figure 1a with the label of the leaf node of the B$^+$-tree of which they are a member (e.g., the block containing Chicago is in leaf node R of the B$^+$-tree).

It is important to observe that the above combination of the bucket PR quadtree and the B$^+$-tree has the property that the tree structure of the partition process of the underlying space has been decoupled from that of the node hierarchy (i.e., the grouping process of the nodes resulting from the partition process) that makes up the tree directory. More precisely, the grouping process is based on proximity in the ordering of the locational codes and on the minimum and maximum capacity of the nodes of the B$^+$-tree. Unfortunately, the resulting structure has the property that the space that is spanned by a leaf node of the B$^+$-tree (i.e., the blocks spanned by it) has an arbitrary shape and, in fact, does not usually correspond to a d-dimensional hyper-rectangle. In particular, the space spanned by a leaf node may have the shape of a staircase (e.g., the leaf blocks in Figure 1a that comprise leaf nodes S and T of the B$^+$-tree in Figure 1c) or may not even be connected in the sense that it corresponds to regions that are not contiguous (e.g., the leaf blocks in Figure 1a that comprise leaf node R of the B$^+$-tree in Figure 1c). Of course, we could take the minimum bounding box of each node of the B$^+$-tree; however, now the problem is that the boxes will do a poor job of filtering due to a poor fit, and also will not usually be disjoint. Similar problems arise for the minimum

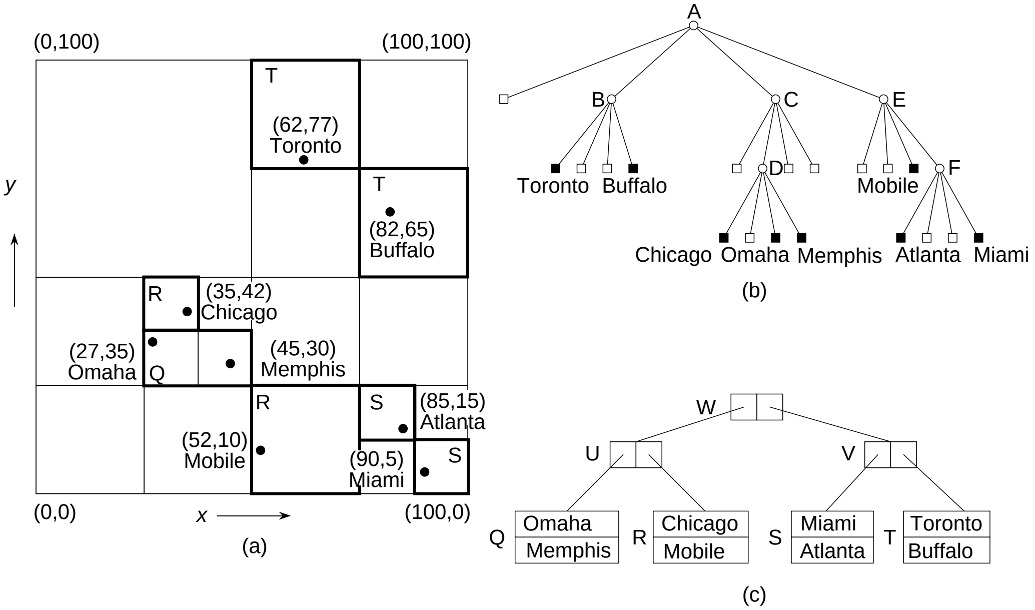


Figure 1: A PR quadtree and the records it represents; (a) the resulting partition of space, (b) the tree representation, and (c) one possible B$^+$-tree for the leaf blocks where each node has a minimum of 2 and a maximum of 3 entries. The nonempty blocks in (a) have been labeled with the name of the B$^+$-tree leaf node in which they are a member.

bounding boxes of the space aggregates corresponding to the nonleaf nodes of the B$^+$-tree. The problem is even more acute if we restrict the minimum bounding box to be one formed by the same rule as the partition process. In this case, the minimum bounding box can be as large as the underlying space (e.g., the minimum bounding quadtree block of leaf node R and nonleaf nodes U, V, and W of the B$^+$-tree in Figure 1c).

An alternative adaptation of the B$^+$-tree is to obtain the buckets in the tree by use of another decomposition process such as a bucket k-d tree. In this case, when using a bucket k-d tree, each block contains a maximum of *b* points and the space is decomposed into hyper-rectangle blocks using the k-d tree [14] decomposition rule which partitions along just one axis at each step instead of on all *d* coordinate axes and does not have to yield congruent blocks at each level of partition. These blocks are the leaf nodes in a B$^+$-tree and serve as buckets. The buckets are aggregated into larger buckets using a k-d tree decomposition rule again. The result is termed a k-d-B-tree [69]. The problem with such a method is that overflow in a leaf node may cause overflow of nodes at shallower depths in the tree whose subsequent partitioning may cause repartitioning at deeper levels in the tree. There are several ways of overcoming the repartitioning problem. One approach is to use the LSD-tree [40] at the cost of poorer storage utilization. An alternative approach is to use representations such as the hB-tree [54, 55, 71] and the BANG file [32] which remove the requirement that each block be a hyper-rectangle at the cost of multiple postings for the hB-tree and nonunique search paths for the BANG file (but see the modified design [36] which overcomes this problem at the loss of a minimal page occupancy guarantee). This has a similar effect as that obtained when decomposing an object into several subobjects in order to overcome the nondisjoint decomposition problem when using an object hierarchy.

In this paper we explore further the use of the principle of decoupling the partition and grouping hierarchies to overcome some of the drawbacks of the tree directory methods discussed above. In particular, we focus on overcoming the following drawbacks:

1. multiple postings in disjoint space decomposition methods,
2. multiple coverage and nondisjointness leading to nonunique search paths in methods employing object hierarchies, and
3. minimum occupancy directory nodes with disjoint, preferably similarly-shaped, hyper-rectangle bounding boxes.

Section 3 discusses the PK-tree [79, 80, 83] and shows how it is used to obtain a space hierarchy where the bounding boxes associated with the nonleaf nodes of the directory are disjoint, congruent with the leaf nodes, and have a known minimum space utilization. The price is that the resulting structure may possibly be unbalanced. The PK-tree achieves this by decoupling the partition process from the grouping process which is based on ensuring that every node has at least *k* objects or blocks. The PK-tree is not plagued by the first two drawbacks as they are inapplicable to it since it is a tree and is a disjoint decomposition. Section 4 discusses the BV-tree [34] and shows how it is used to overcome the multiple repartitioning problems inherent to the k-d-B-tree while having a minimum space utilization and not having the drawbacks of other possible solutions like the LSD-tree [40], the hB-tree [54, 55, 71], and the BANG file [32]. The BV-tree accomplishes this by decoupling the decomposition and directory hierarchies at the cost that the union of the regions associated with the nodes at a given level of the directory does not necessarily contain all of the data points; however, as we will see, this does not effect the efficiency of searching. The BV-tree represents each object just once. In this sense, the BV-tree is similar to an object hierarchy such as the R-tree in that the minimum bounding boxes are not necessarily disjoint nor must they be hyper-rectangles, although the resulting directory hierarchy is not always balanced. However, unlike the R-tree, for a point query, each search is of the same length and visits every level in the decomposition hierarchy thereby ensuring that no node is visited more than once. The BV-tree is not plagued by the third drawback.

3 PK-trees

The PK-tree [79, 80, 83] is a term used to describe a family of data structures based on a disjoint decomposition of the underlying space where the tree structure of the partitioning process is decoupled from the node hierarchy of the grouping process. In our presentation we assume that the decomposition is regular although it need not be so. At a first glance, the PK-tree seems to be similar to bucketing in the sense that the data is grouped on the basis of the number of items that can be stored in a node. The difference is that unlike traditional bucketing methods where there is an upper bound on number of items that can be stored in a node, in the PK-tree there is, instead, a lower bound (as well as an upper bound which is a function of the partitioning method that is used). This results in characterizing the family of data structures as *instantiation methods* with a parameter *k* corresponding to the lower bound. Another important difference is that bucketing is a top-down process which means that for a bucket capacity *k*, the decomposition is halted as soon as a block with *k* or fewer objects is encountered. Thus the bucketing process seeks a maximum enclosing block for the *k* or fewer objects. In contrast, instantiation is a bottom-up process that seeks the minimum enclosing block for the group of at least *k* objects, assuming an instantiation parameter *k*. In other words, the grouping process halts as soon as a block with at least *k* objects is encountered.

The motivation for the development of the PK-tree was to represent high-dimensional point data in applications such as similarity searching, data mining, bioinformatics, etc. In particular, the originators of the PK-tree describe it as a spatial index and evaluate its performance with respect to conventional techniques such as the X-tree [17] and the SR-tree [46]. Unfortunately, as with most indexing methods that rely on a decomposition of the underlying space, performance for high-dimensional data is often unsatisfactory when compared with not using an index at all (e.g., [19]). This is especially true when using uniformly-distributed data where most of the data is found to be at or near the boundary of the space in which it lies [15]. This means that the query region usually overlaps all of the leaf node regions that are created by the decomposition process and a sequential scan appears to be preferable which has led to a number of alternative representations that try to speed up the scan (e.g., VA-file [81], VA$^+$-file [31], IQ-tree [16], etc.). Thus the PK-tree has not found much use and is little known.

In this paper, we focus on the use of the PK-tree for low-dimensional data such as that found in applications such as spatial databases and geographic information systems (GIS). In particular, we discuss the utility of the decoupling of the partitioning and grouping processes that is inherent in the PK-tree and show how it can be used to overcome the drawback associated with tree directory methods of obtaining minimum occupancy directory nodes with disjoint, preferably similarly-shaped, hyper-rectangle bounding boxes. The rest of this section is organized as follows. Section 3.1 defines the PK-tree as well as describes the underlying representation. Section 3.2 describes how searching, insertion, and deletion are implemented in a PK-tree. Section 3.3 concludes the presentation by discussing some of the key properties of the PK-tree.

3.1 Definition

The PK-tree can best be understood by examining an operational definition of its construction given the result of an existing partitioning process. If we view the partitioning process as forming a tree structure (termed a *partition tree*), then, in essence, each PK-tree node represents a subset of the nodes in the partition tree. The PK-tree is constructed by applying a bottom-up grouping process to the nodes of the corresponding partition tree where the partition tree nodes are merged into supernodes until a minimum occupancy, say k, has been attained.

The process starts by creating a PK-tree node for each node in the partition tree. We distinguish between the nodes that are formed for the points (i.e., the leaf blocks of the partition tree some of which are empty) and the nodes formed for the blocks of the partition tree at shallower levels (i.e., the nonleaf blocks), calling them *point nodes* and *directory nodes*, respectively. Next, we start grouping the nodes in bottom-up fashion, using the k-instantiation parameter k to decide how many and which nodes to group. At the deepest level, we eliminate all point nodes whose blocks do not contain a point (i.e., the empty ones), and remove them from their parents (i.e., the links to them). For a directory node a, we check the number of child nodes (which are nonempty) and if a has less than k child nodes, then we remove a and link its children to the parent of a[2]. This process is terminated at the root, which means that the root can have less than k children. The nodes of the partition tree that remain become nodes of the PK-tree and are said to be *k-instantiated* and likewise for the blocks that correspond to them.

As an example, suppose that the partition tree is the PR quadtree given in Figure 1, and let

[2] Note that as a result of this relinking process, the number of entries in a node is almost always greater than k. In Section 3.2 we show that there is indeed an upper bound on the number of entries in a node which depends on k and the branching factor F of the underlying partitioning process.

$k = 3$. We now show how to form the corresponding PK-tree which we call a *PK PR quadtree*. We first find that node B in the PR quadtree has just two nonempty point node children Toronto and Buffalo. Therefore, node B is removed and its children Toronto and Buffalo are linked to B's parent A. Next, we find that directory node D in the PR quadtree has three nonempty point node children Chicago, Omaha, and Memphis, which means that it is k-instantiated and thus D is made a node of the PK-tree. However, D is the only nonempty child of its parent, directory node C of the PR quadtree. Therefore, C is removed and its child D is linked to C's parent A. Checking directory node F in the PR quadtree we find that F has just two nonempty point node children Atlanta and Miami. Therefore, node F is removed and its children Atlanta and Miami are linked to F's parent E. At this point, we have that directory node E in the PR quadtree has three point node children Mobile, Atlanta, and Miami, which means that it is k-instantiated and thus E is made a node of the PK-tree. Note that during this process we removed nodes B, C, and F of the partition tree, while only nodes A, E, and D of the partition tree are also in the PK-tree, besides the nodes corresponding to the data points. The result is that the root node A of the PK-tree has four children corresponding to Toronto, Buffalo, and nodes D and E which are the results of the k-instantiation of Chicago, Omaha, and Memphis, and of Mobile, Atlanta, and Miami, respectively. Figure 2a shows the decomposition of the underlying space while Figure 2b is the corresponding tree structure. The boundaries of the blocks in Figure 2a that correspond to the nonleaf nodes in Figure 2b are drawn using heavy lines. The empty blocks in the first level of the decomposition of the underlying space induced by the PK PR quadtree in Figure 2a are shown shaded. Notice that all of the child blocks are similar but need not be congruent (i.e., they can be of different size as is the case for blocks D and E), nor must the space spanned by their union equal the space spanned by their parent (as is the case for blocks D and E whose union is not equal to that of their parent A).

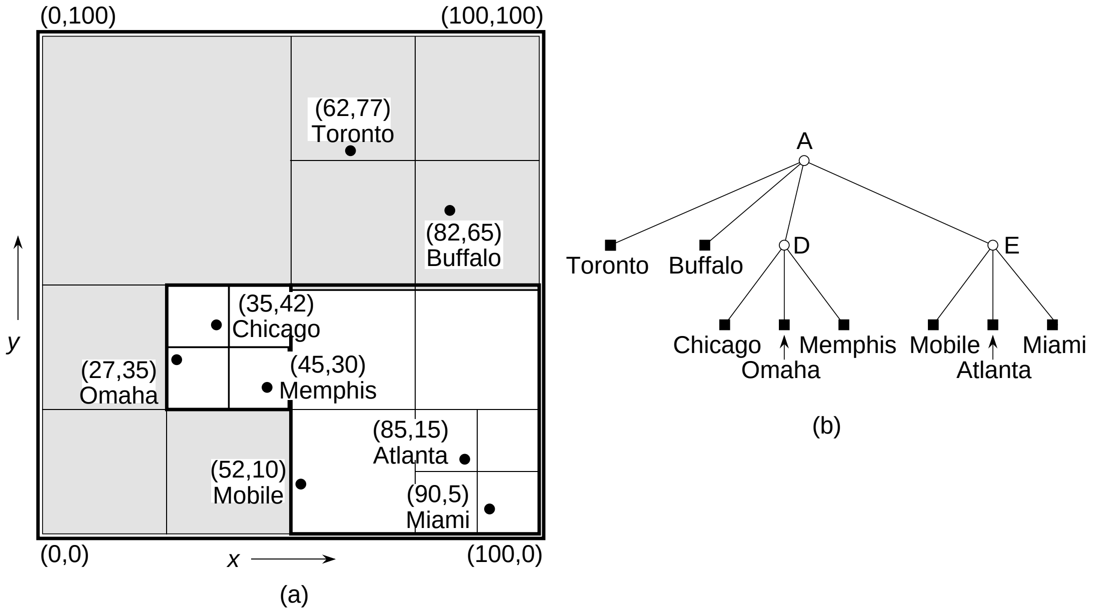


Figure 2: A PK PR quadtree with k=3 corresponding to the data in Figure 1: (a) the resulting partition of space, and (b) the tree representation. The heavy lines indicate the boundaries of the blocks corresponding to the nonleaf nodes in the tree structure. Empty blocks in the first level of decomposition of the underlying space induced by the PK PR quadtree are shown shaded, and are not represented in the tree structure.

It is interesting to observe that when $k = 2$, the process of determining k-instantiated nodes is the same as path compression [6, 60], where nodes with just one child are eliminated thereby linking the child to the parent. In particular, path compression occurs in the PK-tree because empty nodes in the original partition tree are not included in the PK-tree, and thus are not counted in the k or more children that make up each PK-tree nonleaf node. Of course, letting $k = 2$ when the partition tree is a PR k-d tree yields the same result as a path compressed PR k-d tree (which we term a *PK PR k-d tree*).

We now give a more formal definition of the PK-tree. We say that a node or block in the partition tree is *k-instantiated* if it has a corresponding node or block in the PK-tree. We also say that a node or block a in the partition tree *directly contains* a k-instantiated node or block b if b is not contained by any k-instantiated node or block smaller than a. The PK-tree is the hierarchy that consists of just the k-instantiated nodes of the partition tree.

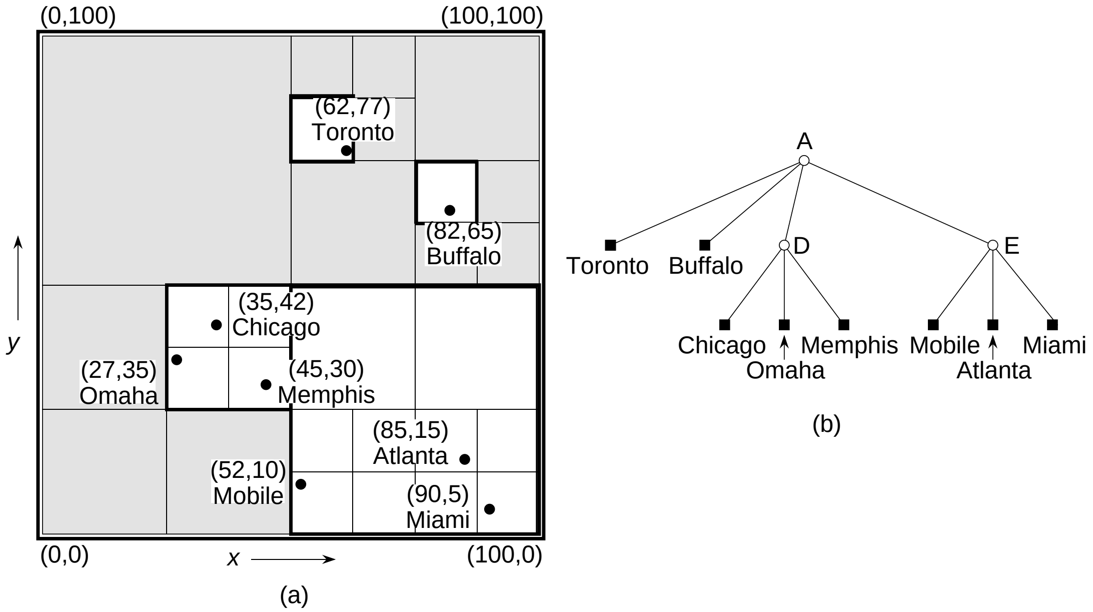
1. A nonempty leaf node (i.e., a point node) in the partition tree is k-instantiated.

2. A nonleaf node (i.e., a directory node) a in the partition tree is k-instantiated if a directly contains at least k k-instantiated nodes or blocks. In other words, if a's corresponding node b in the PK-tree has j ($j \geq k$) children, c_i ($1 \leq i \leq j$), then each node in the partition tree that corresponds to one of c_i is both a descendant of a and is k-instantiated. Furthermore, there are no other k-instantiated nodes in the partition tree on the path from a to c_i.

3. The root node of the partition tree is k-instantiated (i.e., regardless of how many children that its corresponding node in the PK-tree has).

Although this description appears to imply that the PK-tree is a static structure that can be built only once a partition tree has been formed, this is not the case as the PK-tree can also be built dynamically as the corresponding partition tree is updated. In this case, as new data is inserted into the PK-tree, nodes become k-instantiated in a process that is termed *k-instantiation*. Similarly, the deletion of data can cause k-instantiated nodes to cease being k-instantiated in a process that is termed *k-deinstantiation*. Moreover, k-instantiation of a node a often causes k-deinstantiation of the node b that contains a, and vice versa. In fact, as we will see in Section 3.2, dynamic insertion and deletion often result in a cascade of alternating k-instantiations and k-deinstantiations possibly reaching all the way to the root.

The child blocks (i.e., both point nodes and directory nodes) of each node a are represented by their locational codes and are sorted in increasing order. This facilitates searching for a particular point p as we can make use of a binary search for the block in a that contains p. This implementation poses a problem when a is a point node as the coordinate values of the points are usually specified with greater precision than permitted by the resolution of the underlying space. Thus it is usually stipulated that the underlying space has some finite resolution and therefore the locational code of the point node is really the locational code of the 1×1 block that contains the point. In addition, besides the locational code, associated with each entry corresponding to child block b is a pointer to either the actual node in the PK-tree corresponding to b when b is a directory node in the partition tree, or a pointer to a record containing the rest of the information about the point (i.e., its coordinate values and any other relevant information). Of course, if two points fall into the same 1×1 cell, then they have the same locational code. However, they can still be distinguished by the fact that the combination of the locational code and pointer to the rest of the information about the point is unique. It is interesting to observe that the stipulation that the points are represented by 1×1 blocks means that the partition tree in our example is really an MX quadtree (e.g., [74]) instead of a PR quadtree and the corresponding PK-tree is really a PK MX quadtree (see Figure 3). Now, we see that every node in the PK MX quadtree is a minimum enclosing quadtree block of the points in the space spanned by its corresponding block (whereas this was not quite true for the blocks corresponding to the point nodes when the partition tree is a PR quadtree).



(a)

(b)

Figure 3: A PK MX quadtree with k=3 corresponding to the data of Figure 1: (a) the resulting partition of space, and (b) the tree representation. The heavy lines indicate the boundaries of the blocks corresponding to the nodes (leaf and nonleaf) in the first level of the tree structure. Empty blocks in the first level of decomposition of the underlying space induced by the PK MX quadtree are shown shaded, and are not represented in the tree structure.

3.2 Search, Insertion, and Deletion

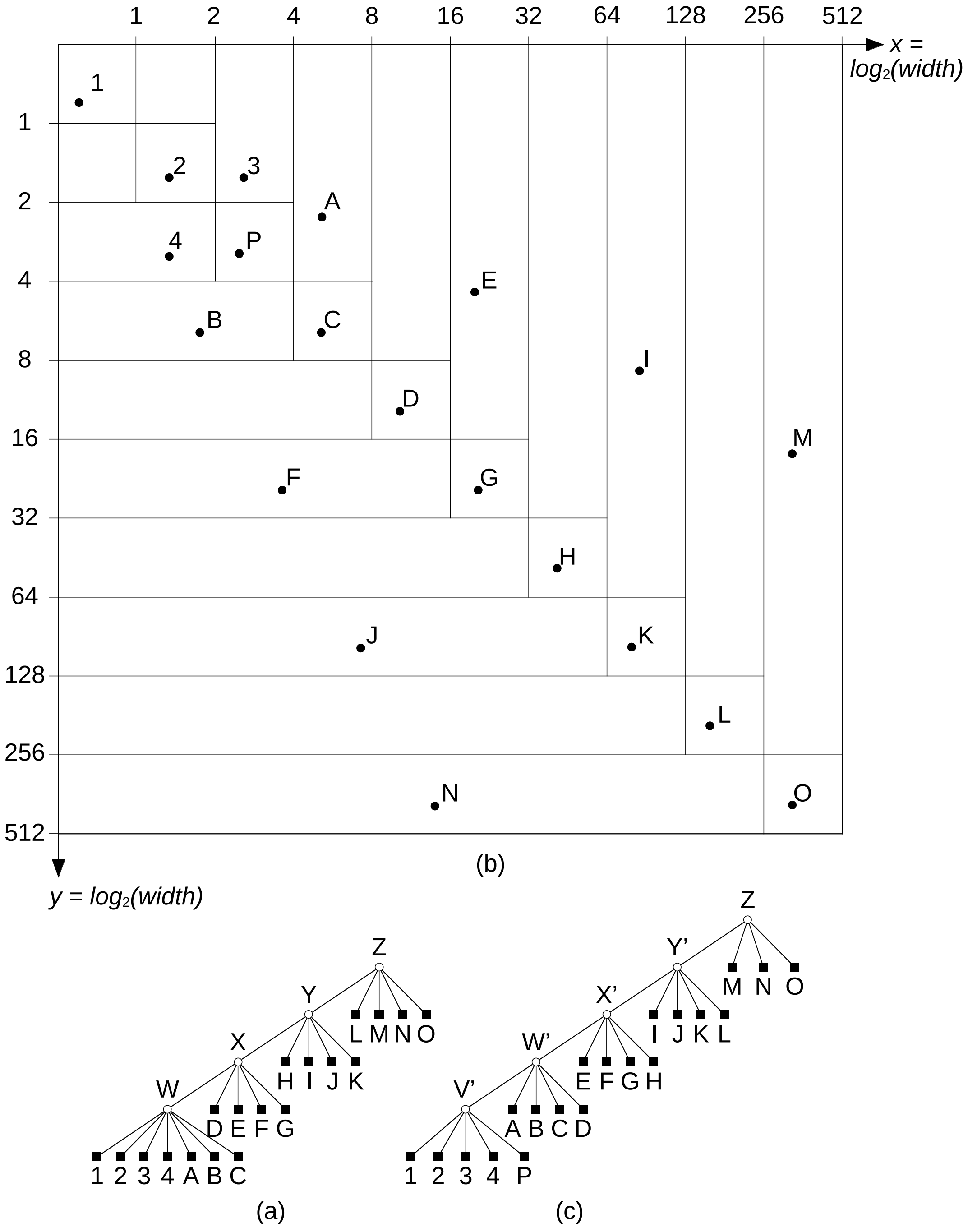
Although our operational definition (i.e., in terms of the *k*-instantiation operation) of the PK-tree implied that we know all of the data items before building it (i.e., it is static), as we pointed out in Section 3.1, the PK-tree is a dynamic structure in the sense that data can be inserted and removed one at a time without having to rebuild the structure from scratch. However, checks for *k*-instantiation and *k*-deinstantiation may have to be applied at each level of the PK-tree during the grouping process. In particular, once the position where the point to be inserted, or the point to be deleted, has been found, we continuously check in a bottom-up manner for the applicability of *k*-instantiation or *k*-deinstantiation until it is no longer applicable, or we have reached the root of the structure. This is a direct consequence of the fact that the PK-tree is defined in a bottom-up manner (i.e., recall that the nodes of the PK-tree are always defined in terms of their descendants).

Before we can start the insertion and deletion process for a point *p*, we must first search for the node whose corresponding block contains *p*. This is done by a top-down process starting at the root of the PK-tree that examines the locational codes of the entries in each node and descends the appropriate child. Once we cannot descend farther, the process is exited after performing the following checks. In the case of insertion, if the corresponding block is empty or is a point node that does not contain p, then the insertion is made. In the case of deletion, if the corresponding block is a point node that contains p, then the deletion is made.

The insertion of a point p, which belongs in a block b of the partition tree which is in turn contained in block c corresponding to nonleaf node q in the PK-tree, may result in the k-instantiation of an ancestor block b' of b in q. In this case, a new node t is created which corresponds to b' whose elements are p and $k-1$ other elements of q. Observe that b' corresponds to the minimum enclosing block of p and the $k-1$ elements of q that form t. Note also that t cannot contain more than k elements as otherwise t would have already existed since the block in the partition tree corresponding to t was not k-instantiated prior to the insertion of p. Node t is made an element of q and t's $k-1$ elements, excluding p, are removed from q. This action may cause the block in the partition tree corresponding to q (i.e., c) to cease being k-instantiated as less than k elements may be left in it (i.e., c is k-deinstantiated). This means that the elements of q must be made elements of the father f of q in the PK-tree. This process is applied recursively on f, with t taking on the role of p, as long as we end up with a node that contains less than k elements or encounter the root at which time we halt. Recall that according to the definition of the PK-tree, the root's corresponding block in the partition tree is always k-instantiated, regardless of the number of nonempty children.

For example, consider the PK PR quadtree with $k = 5$ with points 1–4 and A–O and nonleaf nodes W–Z given in Figure 4a. Figure 4b is the block decomposition of the underlying space of size $2^9 \times 2^9$ induced by the corresponding PR quadtree (i.e., the partition tree) where these points (and an additional point P which will be subsequently inserted) lie. All blocks in Figure 4b are square and the points can lie in any location in the square in which they are found. Note that the block decomposition in Figure 4b is not an MX quadtree since as soon as a block contains just one point, we halt the decomposition, whereas in the MX quadtree we would descend to the final level where each point is a 1×1 block. Of course, since the PK-tree effectively "path compresses" all of the extra decomposition, there is no point in using the MX quadtree. Observe also that the example in Figure 4b is drawn using a logarithmic scale as the widths of the blocks at each successive level of decomposition are one half the widths of the blocks at the immediately preceding level. Figure 4c shows the result of inserting point P into the PK PR quadtree of Figure 4a given the physical interpretation of Figure 4b. Nonleaf nodes V'–Y' have been added during the insertion process, while nodes W–Y (see Figure 4a) have been removed.

Notice that the insertion of P has led to the k-instantiation of the partition tree node corresponding to the minimum enclosing block of it and points 1–4 thereby causing the creation of node V' in Figure 4c. In fact, this act of k-instantiation has caused the k-deinstantiation of the block in the partition tree corresponding to node W, as now it contains only four elements (i.e., A–C and V'), thereby causing the promotion of the partition tree blocks corresponding to points A–C and node V' to node X. This causes the k-instantiation of the minimum enclosing partition tree block of points A–D and node V', and the formation of a new node W'. This action results in the k-deinstantiation of the block in the partition tree corresponding to node X as now it contains only four elements (i.e., points E–G and node W'), thereby causing the promotion of the partition tree blocks corresponding to points E–G and node W' to node Y. This causes the k-instantiation of the minimum enclosing partition tree block of points E–H and node W' and the formation of a new node X'. Once again, this action results in the k-deinstantiation of the block in the partition corresponding to node Y, as now it contains only four elements (i.e., points I–K and node X'), thereby causing the promotion of the partition tree blocks corresponding to points I–K and node X' to node Z. This causes the k-instantiation of the minimum enclosing partition tree block of points I–L and node X' and the formation of a new node Y'. At this point, node Z remains with just four elements corresponding to points M–O and node Y' but as Z is the root of the tree, Z's corresponding block in the partition



(b)

(a)

(c)

Figure 4: (a) Tree representation of an example PK PR quadtree for k=5. (b) A block decomposition of the underlying space (at logarithmic scale but all blocks are square and of a width which is a power of 2) where the labeled points can lie so that the tree representation in (a) would be physically realizable. (c) The result of inserting point P into the PK PR quadtree in (a).

tree is *k*-instantiated by definition. Thus we are done and there is no need to be concerned about whether Z has less than *k* elements.

It is important to note that this example does not represent typical data as it corresponds to data that is extremely skewed. It was chosen as it represents the worst possible case for the insertion algorithm in the sense that it results in a k-instantiation and k-deinstantiation at each level of the PK-tree (for more details, see Section 3.3).

Deletion of a point p which is an element of nonleaf node q in the PK-tree is handled in a similar manner to insertion. The difference from insertion is that deletion may result in the removal of nodes from the PK-tree whose corresponding blocks in the partition tree were formerly k-instantiated. If the block in the partition tree corresponding to q is still k-instantiated after the deletion, then no further action must be taken. Otherwise, q must be deallocated, and its remaining $k-1$ elements are made elements of the father f of q in the PK-tree (i.e., the block in the partition tree corresponding to q is k-deinstantiated). The insertion of additional elements in f may mean that one of the subblocks, say b', of the partition tree block b corresponding to f that contains the newly inserted $k-1$ elements plus at least one other element may become k-instantiated. In this case, a new node t will be created which corresponds to b'. Note that t can contain more than k elements. Node t is made an element of f and t's elements are removed from f. This action may cause the block in the partition tree corresponding to f (i.e., b) to cease being k-instantiated as less than k elements may be left in it (i.e., b is k-deinstantiated). This means that the elements of f must be made elements of the father g of f This process is applied recursively on g, with t taking on the same role as p in the insertion process, as long as we end up with a node that contains less than k elements or encounter the root at which time we halt. Recall that according to the definition of the PK-tree, the root's corresponding block in the partition tree is always k-instantiated, regardless of the number of nonempty children.

For example, consider the PK PR quadtree with $k = 5$ with points 1–4, A–M, and P and nonleaf nodes V–Z in Figure 5a. Figure 5b is the block decomposition of the underlying space of size $2^9 \times 2^9$ induced by the corresponding PR quadtree (i.e., the partition tree) where these points lie. All blocks in Figure 5b are square and the points can lie in any location in the square in which they are found. Observe that the example in Figure 5b is drawn using a logarithmic scale as the widths of the blocks at each successive level of decomposition are one half the widths of the blocks at the immediately preceding level. Figure 5c shows the result of deleting point P from the PK PR quadtree of Figure 5a given the physical interpretation of Figure 5b. Nonleaf nodes V–Z are the nonleaf nodes in the original PK-tree. Nodes V'–X' have been added during the deletion process, while nodes V–Y have been removed.

Notice that the deletion of P has led to the k-deinstantiation of the partition tree node corresponding to V (as it has just four elements corresponding to points 1–4). This action results in the removal of node V in Figure 5c and the promotion of points 1–4 to node W. This causes the k-instantiation of the minimum enclosing partition tree block of the points 1–4 and the block corresponding to point A, and a new node V' is formed which is made an element of node W. This action results in the k-deinstantiation of the block in the partition tree corresponding to node W, as now it contains only four elements (i.e., points B–D and node V'), thereby causing the promotion of the partition tree blocks corresponding to points B–D and node V' to node X. This causes the k-instantiation of the minimum enclosing partition tree block of points B–E and node V', and the formation of a new node W' which is made an element of node X. This results in the k-deinstantiation of the block in the partition tree corresponding to node X, as now it contains only four elements (i.e., points F–H and node W'), thereby causing the promotion of partitioning process blocks corresponding to points F–H and node W' to node Y. This causes the k-instantiation of the minimum enclosing partition tree block of points F–I and node W', and the formation of a new node X' which is made an element of node Y. This results in the k-deinstantiation of the block in the partition tree corresponding to node Y, as now it contains only four elements (i.e., points J–L and node X'), thereby causing the

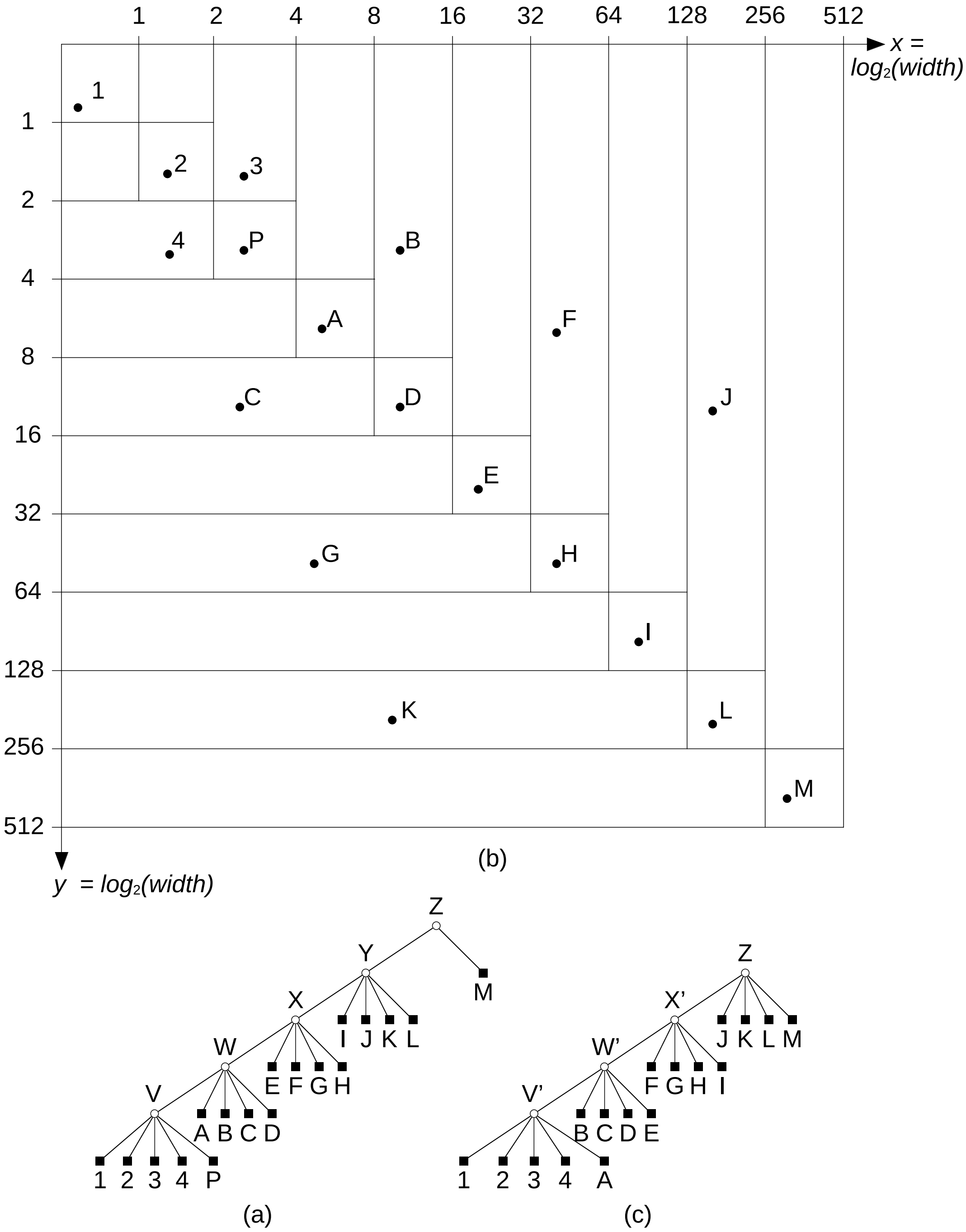


Figure 5: (a) Tree representation of an example PK PR quadtree for k=5. (b) A block decomposition of the underlying space (at logarithmic scale but all blocks are square and of width which is a power of 2) where the labeled points can lie so that the tree representation in (a) would be physically realizable. (c) The result of deleting point P from the PK PR quadtree in (a).

promotion of the partition tree blocks corresponding to points J–L and node X' to node Z which is the root node with just one other element (i.e., M). At this point, node Z has five elements but as

Z is the root of the tree, Z's corresponding block in the partition tree is k-instantiated by definition. Thus we are done and there is no need to be concerned about whether Z has less than k elements.

It is important to note that this example does not represent typical data as it corresponds to data that is extremely skewed. It was chosen as it represents the worst possible case for the deletion algorithm in the sense that it results in a k-instantiation and k-deinstantiation at each level of the PK-tree (for more details, see Section 3.3).

It is interesting to observe that at times a node may temporarily contain more than the maximum possible number of elements, which is $F \cdot (k-1)$ for a partitioning process with a branching factor F. This situation arises when k-deinstantiation takes place (i.e., right after k-instantiation) or when an insertion is made into a PK tree node that is full. This situation can also arise when a point is deleted. To see how it can occur, suppose that node n contains $F \cdot (k-1)$ elements. If we insert new element into n, then n will momentarily contain $F \cdot (k-1)+1$ elements. Similarly, a k-deinstantiation involving n, that replaces one of the elements in n by $k-1$ elements from a child node, results in n containing $F \cdot (k-1)-1+(k-1) = F \cdot (k-1)+k-2 = (F+1) \cdot (k-1)-1$ elements. Of course, in both cases, the newly added elements of n will be immediately k-instantiated with other element(s) of n and the node will once again contain a maximum of $F \cdot (k-1)$ elements.

Note that the physical size of a node (in bytes) is dictated by the chosen page size of the disk resident tree file (often 4KB). Thus, it is undesirable that the node becomes larger than the physical size. There are essentially two ways to overcome this problem. The first is to allow the size of the nodes to exceed the physical size (i.e., the page size) when in memory (e.g., by allocating a larger memory block or by maintaining an overflow buffer of $k-2$ elements, which is used only when a node overflows in the described manner). The second is to modify the code for the PK-tree update operations (i.e., both insertion and deletion algorithms) so that they avoid the overflow problem. In particular, before adding m elements to a node n (either through insertion or k-deinstantiation), we check whether we can k-instantiate these m new elements together with $k-m$ existing elements in n. If this is possible, then we perform the k-instantiation without adding the m elements to n; otherwise, it is safe to add the new elements without exceeding the maximum size.

3.3 Discussion

What makes the PK-tree so attractive is that it is possible to make the directory nodes sufficiently large (i.e., by increasing their maximum capacity) so that each node can be stored efficiently in a disk page. The number of elements comprising a PK-tree node differs on the basis of the underlying partitioning process. For example, in a two-dimensional quadtree implementation such as the PK PR quadtree, each node has a minimum of k children and a maximum of $4 \cdot (k-1)$ children. Thus each node is between 25% and 100% full. Similarly, in a three-dimensional octree implementation, each node has a minimum of k children and a maximum of $8 \cdot (k-1)$ children — that is, each node is between 12.5% and 100% full. Thus the lower the fanout of the partition process, the greater the storage utilization. This increase in the storage utilization is a direct consequence of decoupling the grouping and partitioning processes, and basing the grouping process on instantiation rather than bucketing.

This suggests that the storage utilization is maximized when using a binary tree such as a PR k-d tree (i.e., a PK PR k-d tree). In this case, each node has a minimum of k children and a maximum of $2 \cdot (k-1)$ children regardless of the dimensionality of the underlying space. Thus each node is guaranteed to be at least half full (i.e., between 50% and 100% full). This is the same guarantee that is provided by the classical B-tree definition which is the result of applying a variation of bucketing to a binary search tree. The advantage of using the PK-tree over the method described in Section 2 where the bucket PR quadtree (as well as a bucket PR k-d tree) was combined with a B^+-tree is that in the PK PR k-d tree, the blocks corresponding to the area spanned by the data in the nodes (both leaf and nonleaf) are always similar in the sense that for d-dimensional data, there are d possible hyperrectangular block shapes. Recall from the discussion of Figure 1 in Section 2 that this was not the case for the combination of a bucket PR quadtree (or any other space decomposition method where the blocks are represented by locational codes) and a B^+-tree. Another advantage of the PK-tree is that in the case of nonleaf nodes, the bucket elements can be points (i.e., objects) or pointers to child nodes, rather than being restricted to being just one type. This was seen to yield greater flexibility but at the cost of a possible absence of balance.

Of course, in order for the directory nodes of the PK-tree to have a large capacity, we must choose values of k larger than the branching factor of the corresponding partitioning process (i.e., 2, 4, and 8, for the k-d tree, quadtree and octree, respectively). However, this is not really a problem as it simply means that we aggregate a larger number of nodes at each level of the tree. Note that k-instantiation, which is the principal idea behind the PK tree can be applied to other partitioning processes that are based on a disjoint decomposition of space (preferably a regular decomposition but not required).

The examples in Figures 4 and 5 point out how and when the process of k-instantiation and k-deinstantiation can take place at each level of the PK-tree. Notice that the particular configuration of data in our examples starts with (results in) each nonleaf node, save the root at the end (start) of the insertion (deletion) process, being completely full (i.e., truly k-instantiated) with respect to the given value of k (e.g., $k = 5$ in our example). They represent the worst-case of the PK-tree in the sense that insertion (deletion) of one point has caused the depth of the PK-tree to increase (decrease) by one level. However, it is the maximum possible increase (decrease) in depth due to the insertion (deletion) of a point. This is in contrast to the conventional PR quadtree where the increase (decrease) in depth depends on the minimum separation between the newly added (deleted) point and the closest existing point to it in the tree. Similar examples of the worst case can be constructed for any value of k.

An additional important observation from these examples is that the maximum depth of a PK-tree with N nodes is $O(N)$ and thus the worst-case I/O time complexity to search (as well as update) is $O(N)$, although Wang et al. [79] have shown that under certain assumptions on the probability distribution of the data (including uniformly-distributed data [83]), the expected search cost is $O(\log N)$. Thus we see that the PK-tree does not behave well (from the perspective of being a balanced structure) for skewed distributions of data as in our example. Clearly, for such data distributions, the linear nature of the worst case of tree structures resulting from the use of a regular decomposition process that plagues all decomposition methods which are insensitive to the order of insertion can never be completely avoided. In contrast, the linear nature of the worst case of tree structures resulting from the use of methods that are not based on a regular decomposition process (e.g., the worst case of a binary search tree when the data is processed in sorted order, and likewise for the worst case of the point quadtree) can be avoided by changing the order of insertion. Nevertheless, for the tree structures that result from the use of a regular decomposition process, we at least can avoid the tests which do not lead to differentiation between data which is achieved by the PK-tree and other methods that make use of path compression (e.g., [59, 60, 61]). Moreover, the skewed data distributions are quite rare and, in addition, the relatively large values of k ensure that the resulting trees are relatively shallow.

4 BV-trees

As we pointed out in Section 2, the main drawback of the k-d-B-tree is that the insertion of a split line may cause a recursive application of the splitting process all the way to the point nodes. For example, this is the case when the region corresponding to directory node f in the k-d-B-tree given in Figure 6a is partitioned at $x = 50$ as shown in Figure 6b to yield two new regions whose corresponding directory nodes are f_1 and f_2. This means that regions B, C, and D lie entirely to the left of $x = 50$ and are inserted into f_1, while regions E, F, H, I, and J lie entirely to the right of $x = 50$ and are inserted into f_2. On the other hand, all elements e of f whose regions are intersected by $x = 50$ (i.e., regions A and G) must be split and new regions are created which are inserted into f_1 or f_2 as is appropriate. This additional splitting process is applied recursively to the children of e and terminates upon reaching the appropriate point nodes. The LSD-tree [40] overcomes the repartitioning problem by choosing a partition line that is not intersected by any of the regions in the overflowing directory node at the possible cost of poor storage utilization of the resulting directory nodes.

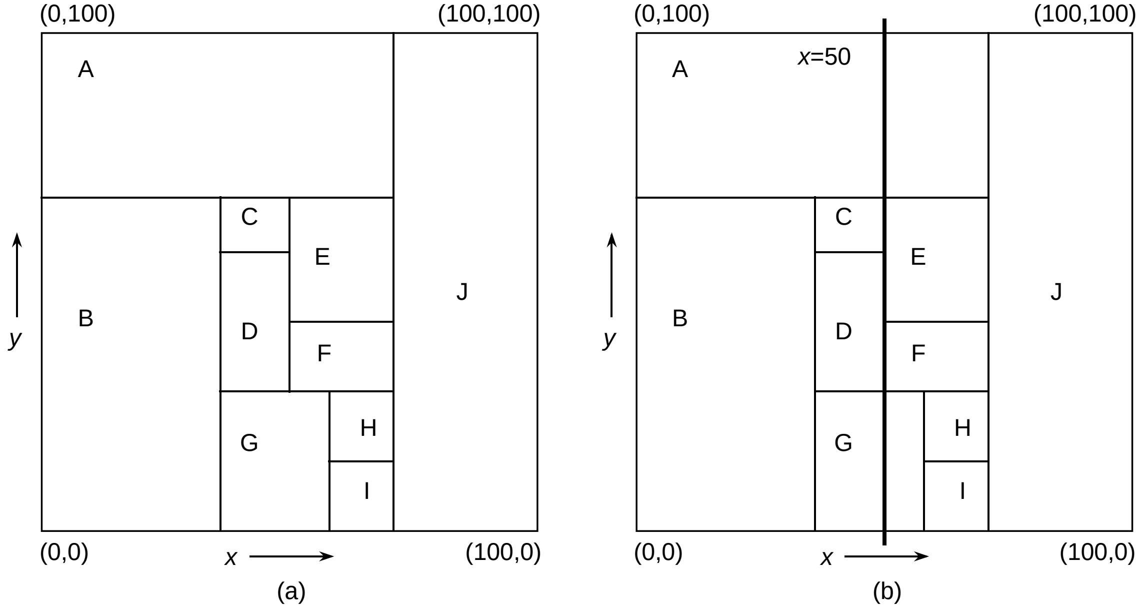


Figure 6: (a) A directory node in a k-d-B-tree with capacity 9 that overflows, and (b) the partition induced by splitting its corresponding region into two regions with corresponding directory nodes at x=50.

The hB-tree [54, 55, 71] and the BANG file [32] avoid the repartitioning problem of the k-d-B-tree as well as the low storage utilization drawback of the LSD-tree by removing the requirement that the portions of the underlying space spanned by the directory nodes resulting from the split must be hyper-rectangles. Instead, both the hB-tree and the BANG file split the region corresponding to each directory node into two regions so that each of their corresponding directory nodes is at least one-third full thereby making the other directory node no more than two-thirds full (the same property also holds for the point nodes). The portions of space spanned by the resulting directory nodes have the shapes of bricks with holes where the holes correspond to directory nodes spanning a hyper-rectangular space which has been extracted from the region corresponding to the directory node that has been split. This analogy serves as the motivation for the name *hB-tree*, where *hB* is an abbreviation for a *holey-brick*. Note that subsequent splits may result in directory nodes that span areas that are not contiguous but this is not a problem [55].

Both the hB-tree and the BANG file employ a directory where the directory nodes are created by a partition process that makes use of variants of a k-d tree. The structure of the hB-tree is such that a particular region may appear more than once in the k-d tree that represents the node. Moreover, in the case of a node split as a result of node overflow, it is possible that some of the nodes in the hB-tree will have more than one parent (called the *multiple posting problem*). This means that the directory in the hB-tree is really a directed acyclic graph (i.e., there are no cycles and the pointers are unidirectional) and thus is not actually a tree. As an example of this situation, consider the region in Figure 7a whose corresponding directory node, say P, is given by the k-d tree in Figure 7b. Suppose that P has overflowed. Of course, this example is too small to be realistic, but it is easy to obtain a more realistic example by introducing additional partition lines into the regions corresponding to A and B in the figure. In this case, we traverse P's k-d tree in a top-down manner descending at each step the subtree with the largest number of leaf nodes (i.e., regions) and stopping as soon as the number of regions in one of the subtrees, say *S*, lies between one-third and two-thirds of the total number of regions in the overflowing directory node *P*. In this example, this is quite easy as all we need to do is descend to the left subtree of the right subtree of the root. The result is the creation of two hB-tree nodes (Figures 7c and 7d) where region A is now split so that it has two parent hB-tree directory nodes and the partition line between these two parent hB-tree directory nodes is posted to the father hB-tree node of P.

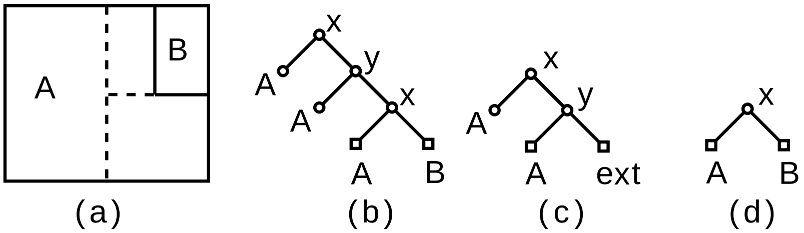


Figure 7: (a) Example region where (b) is the k-d tree of its hB-tree directory node and whose overflow is resolved by partitioning it thereby creating two hB-tree directory nodes having (c) and (d) as their k-d trees. The actual partition of the regions that is induced by the k-d trees is shown by the broken line in (a). The nonleaf nodes of the hB-tree directory nodes are labeled with the name of the key that serves as the discriminator.

If subsequent operations result in further splits of region A, then we may have to post the result of these splits to all of the parent hB-tree directory nodes of A. For example, suppose that region A in Figure 7a is subsequently split into regions A and C as shown in Figure 8a. In this case, the resulting hB-tree nodes are given by Figures 8b and 8c corresponding to Figures 7c and 7d, respectively. Here we see that this could have possibly caused additional node splits to occur. The greater the number of such splits that take place, the more complicated the posting situation becomes. In fact, this complexity was later realized to cause a flaw in the original split and post algorithm for the hB-tree [26, 72] and was subsequently corrected as part of the hB$^{\pi}$-tree [27, 28].

The BANG file differs from the hB-tree in a number of ways. First of all, the BANG file makes use of a regular decomposition to split the underlying space, while the hB-tree splits at arbitrary positions. The BANG file represents each region by a bit string corresponding to the outer boundary of the region that indicates how the outer boundary of the region was created from the original underlying space with alternating bits corresponding to splits along the *x* and *y* axes in the case of

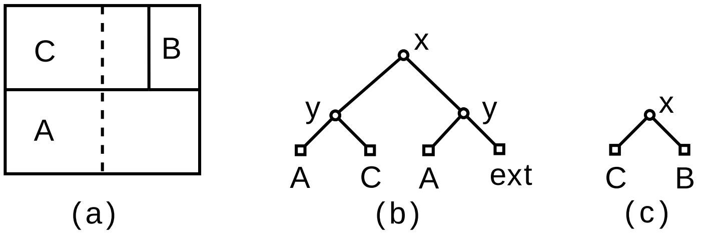


Figure 8: (a) Result of splitting region A in Figure 7a into regions A and C, and the resulting two hB-tree directory nodes having (b) and (c) as their k-d trees which correspond to Figures 7c and 7d, respectively. The actual partition of the regions that is induced by the k-d trees is shown by the broken line in (a). The nonleaf nodes of the hB-tree directory nodes are labeled with the name of the key that serves as the discriminator.

two-dimensional data. These bit strings are stored in some B-tree like structure (see also [66]). The fact that bricks have been extracted from the regions is represented implicitly in the BANG file by ordering the regions in order of increasing size of the outer boundary of their blocks, and search operations are performed in this order. Thus since each region in the BANG file is represented just once, the BANG File [32] does not suffer from the multiple posting problem as it is a tree rather than a directed acyclic graph, However, the use of blocks that are not necessarily hyper-rectangles and the facts that the union of the subspaces within a directory node *a* is not necessarily the same as the space spanned by *a* and the need to examine the smaller regions first means that some of the search paths in the original design of the BANG file may not be unique. This problem is overcome in a modified design of the BANG File [36] at the price of the loss of the minimum occupancy guarantee.

The BV-tree [34] was developed as a means of overcoming the multiple posting problem in the hB-tree and either the nonunique search paths or the lack of a minimum storage occupancy in the BANG file by decoupling the hierarchy inherent to the tree structure of the directory made up of directory nodes from the containment hierarchy associated with the recursive partitioning process of the underlying space from which the data is drawn. The motivation for the BV-tree is to have properties similar to the desirable properties of the B-tree while avoiding the problems described above. In particular, a bounded range of depths (and hence a maximum) can be guaranteed for the BV-tree on the basis of the maximum number of data points and capacity (i.e., fanout) of the point and directory nodes (they may differ). These bounds are based on the assumption that each point and directory node is at least one-third full [34]. Moreover, in spite of the fact that the BV-tree is not quite balanced, search procedures always proceed level to level in the containment hierarchy. For exact match point search, this means that the number of steps to reach a point node is equal to the depth of the containment hierarchy. Note that in this section we use the term *containment hierarchy* instead of *partition hierarchy* because it provides a more accurate description of the constraints on the decomposition process imposed by the BV-tree.

The rest of this section is organized as follows. Section 4.1 defines the BV-tree. Section 4.2 describes how to build a BV-tree and how to search a BV-tree for a region that contains a given point. Deletion is also discussed briefly. Section 4.3 concludes the presentation by reviewing some of the key properties of the BV-tree and compares it with some other representations including the R-tree.

4.1 Definition

In order for the BV-tree technique to be applicable, the regions in the containment hierarchy are required to be disjoint within each level. Furthermore, the boundaries of regions at different levels of the containment hierarchy are also required to be nonintersecting. These requirements hold when the decomposition rules of both the BANG file and the hB-tree are used for the formation of the containment hierarchy. However, in the case of the BANG file, their satisfaction when splitting and merging nodes is almost automatic due to the use of regular decomposition whereas in the case of the hB-tree we need to make use of an explicit k-d tree to represent the history of how the regions were split. In contrast, in the case of the BANG file, the splitting history is implicitly represented by the bit string corresponding to each region which indicates how it was created from the original underlying space.

As in the BANG file and in the hB-tree, there are two ways in which regions in the BV-tree are created that satisfy these properties. In particular, upon node overflow, the underlying space (e.g., Figure 9a) is recursively decomposed into disjoint regions either by splitting them into what we term *distinct regions* (e.g., Figure 9b) or *contained regions* (e.g., Figure 9c). Splitting into contained regions is achieved via the extraction of smaller-sized regions thereby creating one region with holes. The disjointness of the regions at a particular level in the containment hierarchy ensures that the outer boundaries of the regions do not intersect (although they may touch or partially coincide). The nonintersection of the outer boundaries holds for all pairs of regions, and not just for those at the same level of decomposition. The BV-tree is built in a bottom-up manner so that data is inserted in point nodes (of course, the search for the point node to insert into is done in a top-down manner). Overflow occurs when the point node is too full at which time the point node is split thereby leading to the addition of entries to the BV-tree node that points to it which may eventually need to be split when it points to more point nodes than its capacity.

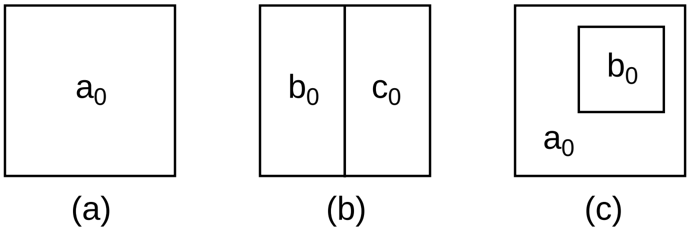


Figure 9: (a) Example region and the result of splitting it into (b) distinct regions or (c) contained regions.

Before proceeding further, let us define some of the naming and numbering conventions that we use in our presentation. For a BV-tree of height h (i.e., with h levels of decomposition or containment), the levels in the containment hierarchy are numbered from 0 to h, where 0 is the deepest level (i.e., corresponding to point nodes). Each level in the containment hierarchy has several regions associated with it that span the regions spanned by every point or directory node in the immediately deeper level in the containment hierarchy. The levels in the directory hierarchy are numbered from 1 to h, with level 1 containing the leaf nodes whose entries are point nodes. This convention enables a node at level v in the directory hierarchy to be (normally) associated with a region at level v in the containment hierarchy. For example, we can view the region representing the entire data space as being at level h in the containment hierarchy, and the root node (at directory level h) also represents the entire data space. It is important to distinguish between a level in the containment hierarchy and a level in the directory hierarchy, since these two hierarchies are decoupled in the BV-tree. In fact, the result of decoupling these two hierarchies is that a node at level v of the containment hierarchy appears as an entry of a node at directory level $v+1$ or shallower.

Nevertheless, the use of the term *level* can be quite confusing. Therefore, in order to differentiate between levels in the containment and directory hierarchies, as well as to reduce the potential for confusion, we frequently distinguish between them by using the terms *clevel* and *dlevel*, respectively. However, we do not use the terms *clevel* or *dlevel* when the term *level* is explicitly qualified by using the full expression *directory level* or *level of the containment hierarchy*.

The containment condition can be restated as stipulating that the interiors of any two regions a and b (regardless of whether their holes are taken into account) are either disjoint (with boundaries that are permitted to partially coincide) or one region is completely contained in the other region (i.e., a is in b or b is in a). From the standpoint of the structure of the directory hierarchy, the containment condition implies that if the region corresponding to directory node n is split on account of n having overflowed into regions a and b such that the area spanned by one of the regions corresponding to a child r of n (which must be redistributed into the directory nodes corresponding to a and b) intersects (i.e., overlaps) both a and b, then the region corresponding to r must completely contain either a or b.

For example, consider the region corresponding to directory node n in Figure 10a which has overflowed on account of having 5 items while its capacity is 4. The region corresponding to n has been split into two regions a and b shown in Figure 10b, which are not disjoint and thus have the property that each of their children is either entirely contained in one of them (i.e., s and t are contained in a and u and v are contained in b), while r which overlaps both a and b in fact contains a in its entirety. In the BV-tree, when this happens, r (and most importantly its corresponding region) will be 'associated' with the resulting directory node (termed its *guard*) whose corresponding region it completely contains (see below for more details about the mechanics of this process). Formally, a region r at level l in the containment hierarchy is said to be a *guard* of region a at level m in the containment hierarchy if and only if r is at a deeper level in the containment hierarchy than a (i.e., $l < m$) and r is the smallest of all regions at level l that enclose a. This is shown in our example by associating r with the entry corresponding to a in the replacement directory node n in Figure 10c where it is placed in front of a.

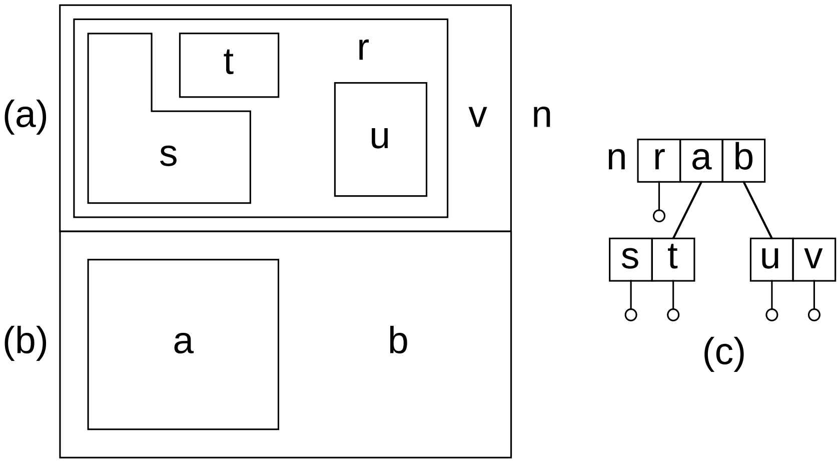


Figure 10: Example illustrating the containment condition when (a) directory node n overflows, (b) the result of splitting it into two directory nodes a and b, and (c) the resulting directory hierarchy.

The BV-tree definition does not stipulate how the underlying space spanned by the elements of the directory structure (i.e., directory nodes) is to be decomposed nor does it restrict the shapes of the resulting regions, provided that their outer boundaries do not intersect. In particular, the underlying space may be decomposed into regions of arbitrary shape and size — that is, neither the space spanned by the regions need be hyper-rectangular nor, as is the case for the hB-tree and BANG file, must the faces of the regions be parallel to the coordinate axes. However, since the space decomposition generated by the BANG file satisfies the nonintersecting outer boundary requirement, the BV-tree method may be used to obtain a variant of the BANG file (i.e., a regular decomposition of the underlying space) that does not suffer from either a nonunique search path or the lack of a minimum storage occupancy.

The fact that regions created at certain levels (i.e., clevels) of decomposition of the underlying space may be associated with directory nodes at shallower directory levels in the directory hierarchy means that the union of the regions associated with all of the directory nodes found at directory level i of the directory hierarchy of the BV-tree does not necessarily contain all of the data points. This is not a problem for searching operations because in such a case each directory node also contains entries that correspond to regions that were not split at deeper levels of the containment hierarchy. These node entries are present at the shallower directory level because otherwise they and their corresponding regions would have had to be split when the directory node corresponding to the region in which they were contained was split. Instead, they were promoted to the next shallower directory level along with the region corresponding to the newly-created directory node that they contained. The directory hierarchy entries whose corresponding regions were created at deeper levels of the containment hierarchy (i.e., the ones that were promoted) serve as *guards* of the shallower directory level directory nodes and are carried down the directory hierarchy as the search takes place.

Since promotion can take place at each directory level, a situation could arise where guards are promoted from dlevel to dlevel. For example, suppose that directory node a has been promoted from being an entry in a node at directory level i to being an entry in directory node x at directory level $i+1$ where a serves as a guard of an entry b which is a directory node. Next, suppose that x overflows, such that its region c is split into two regions whose corresponding directory nodes are d and e which replace x in the parent node y of x. Now, it is possible that the region corresponding to b contains the regions corresponding to either d or e, say e. In this case, b is promoted to y at directory level $i+2$ where it serves as a guard of e, and thus a must also be promoted to y as a is a guard of b. Subsequent splits can result in even more promotion for a. Thus we see that every unpromoted entry in a directory node at directory level i can have as many as $i-1$ guards where we once again assume that the deepest directory nodes are at dlevel 1 (but they contain entries that correspond to point nodes at clevel 0).

4.2 Insertion, Deletion, and Search

The best way to see how the BV-tree works is to consider an example of how it is built. As mentioned above, each directory node a corresponds to a region in the underlying space. The entries in a contain regions and pointers to their corresponding point node or directory node (i.e., entries with regions at clevel 0 correspond to point nodes but regions at shallower clevels correspond to directory nodes). When a is at level i of the directory hierarchy, its entries may be at levels 0 to $i-1$ of the containment hierarchy. The entries whose corresponding regions are at clevels deeper than $i-1$ have been *promoted* to the node a which is at a shallower dlevel in the directory where they serve as *guards* of some of the entries in a. Each entry in a also contains information about the clevel of the region it contains, so that we may distinguish between promoted and unpromoted entries.

Node *a* contains a maximum of *m* entries where *m* is the fanout of *a*. Node *a* is split if it overflows — that is, if the total number of entries in *a* (not distinguishing between promoted and unpromoted entries) exceeds *m*. In our example, we assume a fanout value of seven for the point nodes and four for the directory nodes. When a directory node is split, the BV-tree definition [34] claims that it should be possible to distribute its entries in the newly-created directory nodes *a* and *b* so that each of *a* and *b* is at least one-third full as this is one of the properties of the BV-tree that enables it to guarantee an upper bound on the maximum depth of the directory hierarchy. As we will see below, this may not always be possible for all fanout values thereby requiring some adjustments in the definition of the fanout (see Section 4.3).

Let us start with a set of eight points, labeled 1–8. This means that the capacity (i.e., fanout) of the corresponding point node is exceeded and thus it is split into two regions thereby creating a_0 and b_0, containing points 1–4 and 5–8, respectively, and having a containment and directory hierarchy given in Figures 11a and 11b, respectively.

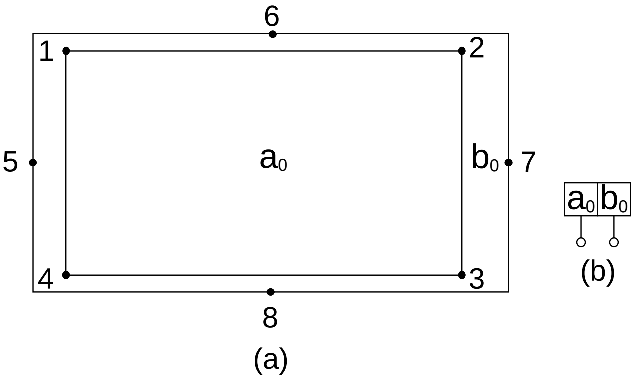


Figure 11: Initial step in creating a BV-tree. (a) Containment hierarchy. (b) Directory hierarchy.

Next, we add points 9–12 which cause a_0 to be split into a_0 and c_0, points 13–16 which cause a_0 to be split into a_0 and d_0, and points 17–20 which cause a_0 to be split into a_0 and e_0, as shown in Figure 12a. However, this causes the single directory node to overflow and it is split into two nodes whose corresponding regions are a_1 and b_1 as shown in Figure 12b. Since region a_0 intersects both a_1 and b_1, while completely containing a_1, a_0 is promoted to the newly-created directory node where it serves as a guard on region a_1. The resulting directory hierarchy is shown in Figure 12c. Note that a_0 is the region which, if it were not promoted, we would have to split along the boundary of a_1. By avoiding this split, we are also avoiding the possibility of further splits at deeper levels of the directory hierarchy. Thus the fact that a_0 appears at this shallower level in the directory hierarchy ensures that a_0 will be visited during any search of the BV-tree for a point on either side of the branch that would have been made if we would have split a_0 which we would have had to do if we did not promote it.

Now, suppose that more points are added so that the underlying space is split several times. We do not go into as much detail as before in our explanation except to indicate one possible sequence of splits. We also no longer show the points in the point nodes.

1. Split d_0 into d_0, f_0, and g_0, which is followed by splitting d_0 into d_0 and h_0. This results in the directory hierarchy given in Figure 13a where the directory node corresponding to a_1, labeled

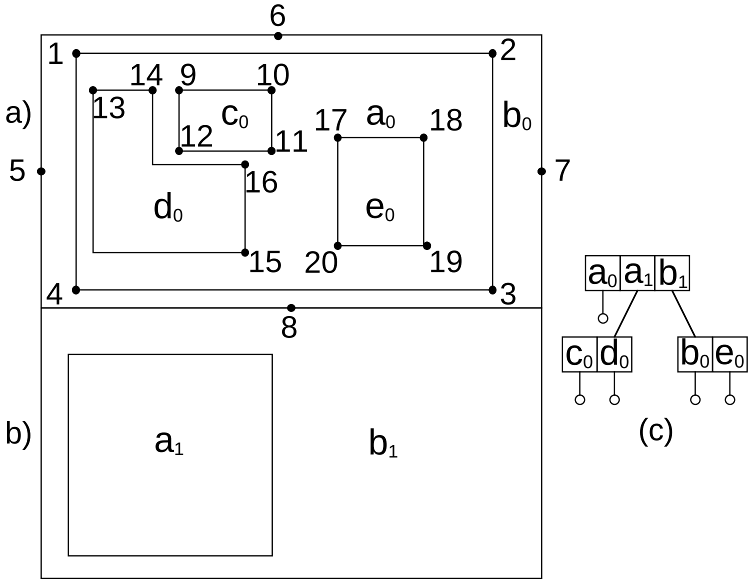


Figure 12: Second step in creating a BV-tree. (a) Containment hierarchy at the deepest level (clevel=0). (b) Containment hierarchy at the shallowest level (clevel=1). (c) Directory hierarchy.

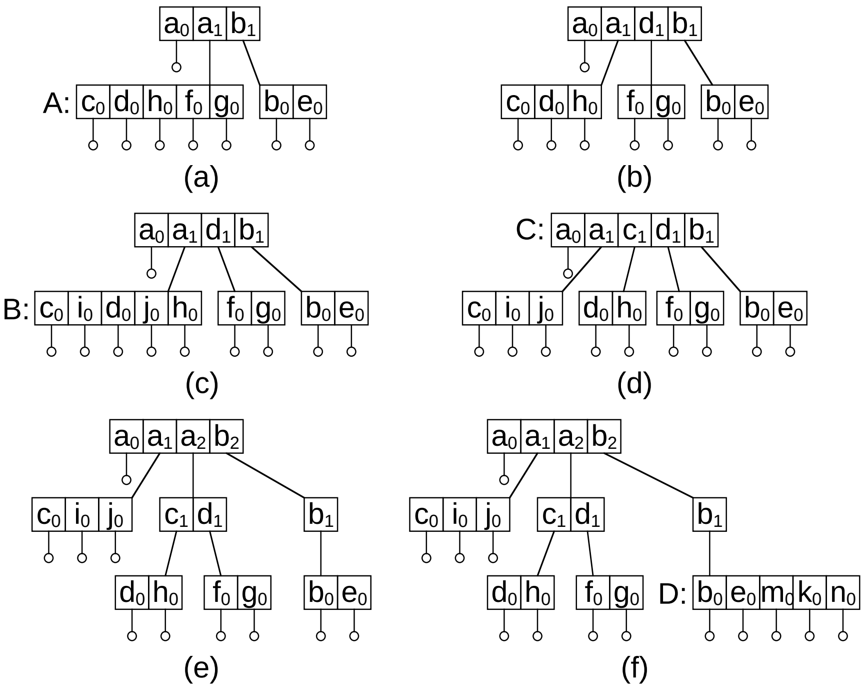


Figure 13: Sequence of intermediate directory hierarchies illustrating the node splitting operations taking place when points are added to the BV-tree of Figure 11 until obtaining the BV-tree of Figure 14.

A, must be split as it overflows. Node A is split into two directory nodes whose corresponding regions are a_1 and d_1 resulting in the directory hierarchy given in Figure 13b.

2. Split c_0 into c_0 and i_0, which is followed by splitting d_0 into d_0 and j_0. This results in the directory hierarchy given in Figure 13c where the directory node corresponding to a_1, labeled B, must be split as it overflows. Node B is split into two directory nodes whose corresponding regions are a_1 and c_1 resulting in the directory hierarchy given in Figure 13d where the directory node corresponding to the root, labeled C, must be split as it overflows.

3. Node C is split into two directory nodes whose corresponding regions are a_2 and b_2 resulting in the directory hierarchy given in Figure 13e. Several items are worthy of note.

   (a) The directory node corresponding to the root also contains an entry corresponding to region a_1 as a_1 intersects both a_2 and b_2, while completely containing a_2. Thus a_1 becomes a guard of a_2.

   (b) Since a_0 originally guarded a_1, a_0 continues to do so, and we find that a_0 is also promoted to the root of the directory hierarchy.

   (c) The directory node corresponding to region b_2 contains only the single region b_1. This means that our requirement that each directory node be at least one-third full is violated. This cannot be avoided in this example and is a direct result of the promotion of guards. This situation arises primarily at shallow dlevels when the fanout value is low compared to the height of the directory hierarchy. It is usually avoided by increasing the value of the fanout as described in Section 4.3.

4. Split e_0 into e_0, k_0, and m_0, which is followed by splitting a_0 into a_0 and n_0. This results in the directory hierarchy given in Figure 13f where the directory node corresponding to b_1, labeled D, must be split as it overflows. There are several ways to split D. One possibility, and the one we describe, is to split D into two directory nodes whose corresponding regions are b_1 and e_1 resulting in the directory hierarchy given in Figure 14d. Since region e_0 intersects both e_1 and b_1, while completely containing e_1, e_0 is promoted to the directory node containing entries corresponding to b_1 and e_1 where e_0 serves as a guard on region e_1. The three levels that make up the final containment hierarchy are shown in Figure 14a–c. Notice that we could have also split D so that the region e_1 contains regions e_0, k_0, and m_0. In this case, e_0 would not be needed as a guard in the directory node containing entries corresponding to regions b_1 and e_1.

Deletion is more complex than insertion. The main issue is how to redistribute the contents of the result of merging an underflowing directory node with another directory node (as well as how to choose it) while adhering to the minimum node occupancy conditions. The redistribution question is one of finding an appropriate splitting strategy for the merged directory nodes. See [33] for more details.

Having shown how to construct the BV-tree, we now examine the process of searching a BV-tree for the region that contains a point p (termed an *exact-match search* here). The key idea behind the implementation of searching in the BV-tree is to perform the search on the containment hierarchy rather than on the directory hierarchy. At a first glance, this appears to be problematic as the BV-tree represents only the directory hierarchy explicitly, while the containment hierarchy is represented only implicitly. However, this is not a problem at all as the containment hierarchy can be, and is, reconstructed on-the-fly as necessary with the aid of the guards when performing the search. In particular, at each stage of the search, the relevant guards are determined and they are carried down the directory hierarchy as the search proceeds. Note that, as we show below, all exact-match searches are of the same length and always visit every level in the containment hierarchy.

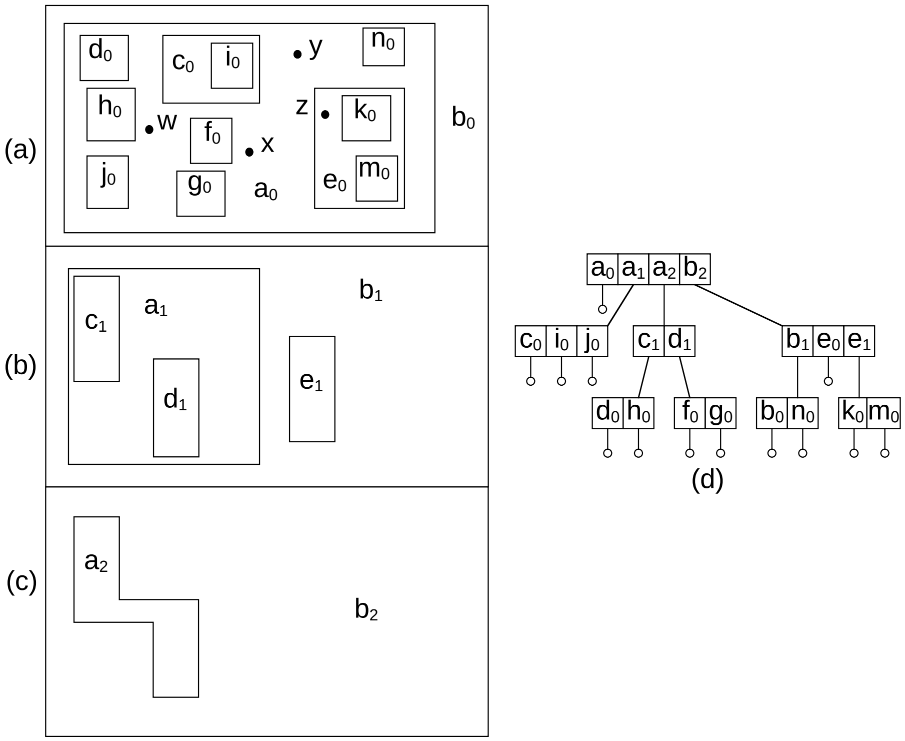


Figure 14: Final step in creating a BV-tree. (a) Containment hierarchy at the deepest level (clevel=0). (b) Containment hierarchy at the intermediate level (clevel=2). (c) Containment hierarchy at the shallowest level (clevel=3). (d) BV-tree.

We explain the search process in greater detail below. Initially, we differentiate between the search depending on whether the directory node being searched is the root of the directory hierarchy, assumed to be at dlevel v. Recall that all unpromoted entries in the root correspond to regions at level $v-1$ in the containment hierarchy. The search starts at the root of the directory hierarchy at dlevel v (assuming that the deepest directory node is at dlevel 1 and describes regions at level 0 in the containment hierarchy). It examines the regions corresponding to all of the entries (i.e., including the guards) and selects the smallest region r (corresponding to entry e_r) that contains the query point p. Entry e_r is termed the *best match*. The level $v-1$ directory node at which the search is continued depends on whether e_r is a guard.

1. Entry e_r is not a guard: Continue the search at the directory node pointed at by e_r with a guard set formed by the current set of guards of e_r (as many as $v-1$ guards, one for each of levels 0 through $v-2$ in the containment hierarchy).

2. Entry e_r is a guard: Find the smallest region s containing r whose corresponding entry e_s is not a guard. It is easy to show that e_s must exist. The area spanned by s may also contain other regions, say the set O, corresponding to guards whose regions contain r. Continue the search at the directory node pointed at by e_s with a guard set consisting of e_r, the set of entries corresponding to the regions that comprise O, and the guards that correspond to regions that contain s. The guards that correspond to regions that contain s are from levels in the containment hierarchy for which O does not contain guards. For each of these clevels,

choose the guard which has the smallest region that contains p. In other words, the guard set can contain at most one guard for each of levels 0 through $v-2$ in the containment hierarchy, for maximum of $v-1$ guards.

Searching at directory node a at dlevel u other than the root uses a similar procedure as at the root. Again, we wish to determine the level $u-1$ directory node at which the search is to be continued. This is done by finding the smallest region r (termed the *best match*) containing p. The difference is that the best match is obtained by examining the guard g at clevel $u-1$ (if it exists) of the u guards at clevels 0 through $u-1$ that comprise the guard set that was brought down from the previous search (i.e., from dlevel $u+1$) and the entries in a that are not guards (note that it may be the case that none of the entries in a are guards as can be seen by examining point w and the directory node whose corresponding region is a_2 in Figure 14a). There are two options:

1. If the best match is the region corresponding to g, then continue the search at the level $u-1$ directory node pointed at by g. The guard set must also be updated as described below.

2. Otherwise, the best match is r, which corresponds to an entry in a (say e_r), and continue the search at dlevel $u-1$ at the directory node pointed at by e_r. Use a guard set obtained by merging the matching guards found at dlevel u with those brought down from dlevel $u+1$. Discard g, the clevel $u-1$ guard brought down from dlevel $u+1$ (if it exists), as g cannot be a part of a directory node at dlevel $u-1$ since each entry at dlevel i of the directory hierarchy can contain at most $i-1$ guards, and the search will next be on regions at] clevel $u-2$. Note that two guards at a given dlevel are merged by discarding the one that provides the poorer match.

Interestingly, there is really no need to differentiate between the treatment of root nodes from that of nonroot nodes. In particular, in both cases, only the unpromoted nodes (i.e., entries whose corresponding regions are not guards) need to be examined. In the case of the root node, initially no guard set is brought down from the previous dlevel. Thus it is initially empty. In this case, the new guard set is formed by finding the guard for each clevel that has the smallest region that contains p.

In order to understand the exact-match search algorithm better, let us see how we find the regions in the BV-tree of Figure 14a that contain the points w, x, y, and z which are a_0, a_0, a_0, and e_0, respectively. Notice that each region has been labeled with subscripts denoting its level in the containment hierarchy. This is also two less than the deepest level in the directory hierarchy at which it can serve as a guard, and one less than the deepest level in the directory hierarchy at which it will get carried down in the guard set.

1. The search for w first examines the root, at dlevel 3, and determines a_2 to be the best match. The search continues at the dlevel 2 directory node whose corresponding region is a_2 (i.e., containing c_1 and d_1) with a guard set consisting of a_0 and a_1. There is no match in the entries in a_2 (i.e., neither c_1 nor d_1 contain w) and thus we use the guard a_1 brought down from dlevel 3 as the best match. The search continues at the dlevel 1 directory node whose corresponding region is a_1 (i.e., containing c_0, i_0, and j_0) with a guard set consisting of a_0. Again, there is no best match in the entries in a_1 (i.e., neither c_0 nor i_0 nor j_0 contain w) but the guard a_0 contains w, and, since the region corresponding to a_0 is at level 0 of the containment hierarchy, we are done. Therefore, w is in a_0.

2. The search for x first examines the root, at dlevel 3, and determines a_1 to be the best match. However, a_1 is a guard and thus it finds the smallest nonguard region containing a_1 which

is b_2. The search continues at the dlevel 2 directory node whose corresponding region is b_2 (i.e., containing b_1, e_0, and e_1) with a guard set consisting of a_0 and a_1. The best match in this node is b_1. However, the clevel 1 guard in the guard set is a_1 and its corresponding region is a better match. The search continues in the dlevel 1 directory node whose corresponding region is a_1 (i.e., containing c_0, i_0, and j_o) with a guard set consisting of a_0. There is no best match in this node but the guard a_0 contains x, and, since the region corresponding to a_0 is at level 0 of the containment hierarchy, we are done. Therefore, x is in a_0.

3. The search for y first examines the root, at dlevel 3, and determines a_0 to be the best match. However, a_0 is a guard and thus it finds the smallest nonguard region containing a_0 which is b_2. The search continues at the dlevel 2 directory node whose corresponding region is b_2 (i.e., containing b_1, e_0, and e_1) with a guard set consisting of a_0. The best match in this node is b_1. There is no clevel 1 guard in the guard set and the search continues at the dlevel 1 directory node whose corresponding region is b_1 (i.e., containing b_0 and n_0) with a guard set consisting of a_0 (we ignore e_0 as it does not contain y). There is no best match in b_1 but the guard a_0 contains y, and, since the region corresponding to a_0 is at level 0 of the containment hierarchy, we are done. Therefore, y is in a_0.

4. The only difference in searching for z from searching for y is that the search is continued at the directory node whose corresponding region is b_1 (i.e., containing b_0 and n_0) with a guard set consisting of e_0 as it provides the better match (i.e., both e_0 and a_0 contain z but a_0 contains e_0). Therefore, z is in e_0.

It is easy to see that all exact-match searches in a BV-tree have the same length and always visit every level in the containment hierarchy. In particular, assume that the root of the directory hierarchy is at level v of the containment hierarchy and that the deepest directory node is at level 1 of the containment hierarchy. The search starts at level v of the containment hierarchy. At each step, the search always descends to a node at the next lower level of the containment hierarchy. This is done either through an unpromoted entry in the current node, or an entry in the guard set representing a region at the next deeper level of the containment hierarchy. The search terminates upon reaching a point node (corresponding to a region at level 0 in the containment hierarchy). Thus all searches are of length v.

4.3 Discussion

One of the main reasons for the attractiveness of the BV-tree is its guaranteed performance for executing exact-match queries, its storage requirements in terms of the maximum number of point and directory nodes, and the maximum depth of the directory structure that are necessary for a particular volume of data given the fanout of each directory node. These guarantees are based in part on the assumption that each point node is at least one-third full [34]. This is justified by appealing to the proof given for the hB-tree's satisfaction of this property [55] (and likewise for a variant of the BANG file where the search paths are not unique). However, the presence of the guards complicates the use of the assumption that each directory node is at least one-third full (which holds for the hB-tree and for the variant of the BANG file where the search paths are not unique). For example, while illustrating the mechanics of the BV-tree insertion process, we saw an instance of a directory node split operation where the nodes resulting from the split were not at least one-third full (recall Figure 13e). This situation arose because of the presence of guards whose number was taken into account in the node occupancy when determining if a directory node overflows. However, the guards are not necessarily redistributed among the nodes resulting from the split. Instead, they are often promoted thereby precipitating the overflow of directory nodes at shallower depths of the directory hierarchy.

At this point, let us re-examine the situation that arises when a directory node overflows. There are two cases depending on whether the overflowing directory node is a guard node. When the overflowing directory node is not a guard node, the problem described earlier can be alleviated by choosing F to be large. An alternative method of avoiding this problem altogether is to permit the directory nodes to be of variable size and to take only the number of unpromoted entries into account when determining how to split an overflowing node.

When the overflowing directory node is a guard node, we must take into account the question of whether the two resulting nodes satisfy the same criteria for promotion as the original overflowing node [35]. If yes, then the situation is quite simple as the corresponding entry in the directory node at the next shallower depth is replaced by entries for the two new guard nodes, and the overflow continues to be propagated, if necessary. Otherwise, the situation is more complex as it may involve demotion which may have an effect on the rest of the structure. Freeston [35] suggests that the demotion be postponed at the cost of requiring larger directory nodes. The exact details of this process are beyond the scope of our discussion.

The implementation of the exact-match search process demonstrates the utility of decoupling the directory hierarchy from the containment hierarchy. Although the BV-tree was designed to overcome the repartitioning and multiple posting problems associated with variants of the k-d-B-tree and the hB-tree, the BV-tree can also be viewed as a special case of the R-tree which tries to overcome the shortcomings of the R-tree. In this case, the BV-tree is not used in its full generality since in order for the analogy to the R-tree to hold, all regions must be hyper-rectangles.

Recall that the main drawback of the k-d-B-tree and its variants is the fact that region splits may be propagated downwards. This is a direct result of the fact that regions must be split into disjoint subregions where one subregion cannot contain another subregion. The hB-tree attempts to overcome this problem but it suffers from the multiple posting problem. In this case, instead of splitting point nodes into several pieces, directory nodes are referenced several times. In terms of efficiency of the search process, the multiple posting problem in the hB-tree is analogous to the multiple coverage problem of the R-tree. In particular, the multiple coverage problem of the R-tree is that the area containing a specific point may be spanned by several R-tree nodes since the decomposition of the underlying space is not disjoint. Thus just because a point was not found in the search of one path in the tree, does not mean that it would not be found in the search of another path in the tree. This makes search in an R-tree somewhat inefficient.

At a first glance, the BV-tree suffers from the same multiple coverage problem as the R-tree. This is true when we examine the directory hierarchy of the BV-tree. However, the fact that the search process in the BV-tree carries the guards with it as the tree is descended ensures that only one path is followed in any search, thereby compensating for the multiple coverage. Notice that what is really taking place is that the search proceeds by levels in the containment hierarchy, even though it may appear to be backtracking in the directory hierarchy. For example, when searching for point x in the BV-tree of Figure 14a, we immediately make use of the guards a_0 and a_1 in our search of region b_2. We see that these three regions have quite a bit in common. However, as we descend the BV-tree, we find that some of the regions are eliminated from consideration resulting in the pursuit of just one path.

The use of the containment hierarchy in the BV-tree can be thought of as leading to a special case of an R-tree in the sense that the containment hierarchy serves as a constraint on the relationship between the regions that form the directory structure. For example, in the R-tree, the regions are usually hyper-rectangles (termed *bounding hyper-rectangles*), whereas in the BV-tree they can be of any shape. In particular, the constraint is that any pair of bounding hyper-rectangles of two children *a* and *b* of an R-tree node *r* must be either disjoint or one child is completely contained in the other child (i.e., *a* is in *b* or *b* is in *a*). In addition, if we were to build an R-tree using the BV-tree rules, then we would have to modify the rules as to the action to take when inserting a point (or, more generally, an object) that does not lie in the areas spanned by any of the existing bounding hyper-rectangles. In particular, we must make sure that the expanded region does not violate the containment hierarchy requirements. Overflow must also be handled somewhat differently to ensure the promotion of guards so that we can avoid the multiple coverage problem. Of course, once this is done, the structure no longer satisfies the property that all leaf nodes are at the same depth. Thus the result is somewhat like the S-tree [4] which is a variant of the R-tree designed to deal with skewed data.

It is important to note that if the underlying objects are not points, then it may be impossible to guarantee satisfaction of the containment hierarchy requirements when the regions that form the elements of the directory structure are bounding hyper-rectangles. In fact, an example of such an impossible case can be constructed when the minimum bounding objects of the underlying objects are hyper-rectangles [5]. In this case, we may need to use more general and application-specific bounding structures, and most likely also require that the objects do not overlap. The investigation of the exact relationship between the R-tree and the BV-tree is an issue for further study.

5 Concluding Remarks

The PK-tree and the BV-tree show the utility of decoupling the partitioning and grouping processes that form the basis of most spatial indexing methods that are based on the use of tree directories thereby overcoming the following drawbacks:

1. Multiple postings in disjoint space decomposition methods that lead to balanced trees such as the hB-tree where a node split may be such that one of the children of the node that was split becomes a child of both of the nodes resulting from the split.

2. Multiple coverage and nondisjointness of methods based on object hierarchies such as the R-tree which lead to nonunique search paths.

3. Directory nodes with similarly-shaped hyper-rectangle bounding boxes with minimum occupancy in disjoint space decomposition methods such as those based on quadtrees and k-d trees that make use of regular decomposition.

The BV-tree addresses the first two drawbacks while the PK-tree addresses the last drawback. Of course, this solution does come at a cost which is that we can no longer guarantee that the resulting structure is balanced. However, since the nodes have a relatively large fanout, the deviation from a balanced structure is relatively small — often just one level in the BV-tree (see [34]). The PK-tree is conceptually much simpler than the BV-tree. In part, this is because the PK-tree directly represents the decomposition hierarchy. Also, traversal of the tree is simple, whereas in the BV-tree, we must take guard nodes into account. Nevertheless, the BV-tree has the advantage that its height is guaranteed to be logarithmic in the number of data items, while as we pointed out in Section 3.3, at best, only an average case argument can be made for the PK-tree. Thus all searches take the same amount of time in the BV-tree while this is not necessarily the case in the PK-tree.

It is important to note that this paper should not be interpreted as a comparison of the PK-tree and the BV-tree spatial indexing methods. Such a comparison is not really possible as these methods are quite different as the PK-tree is based on a decomposition of the underlying space that makes use of a regular decomposition while the BV-tree is based on an object hierarchy. Although the BV-tree does overcome the drawbacks associated with yielding a possibly nondisjoint decomposition of space and is good at being able to distinguish between occupied and unoccupied space for a particular data set, the BV-tree cannot correlate occupied space in two different data sets. In other words, for the BV-tree, the bounding boxes of objects or collections of objects in the two data sets are not in registration which means that more intersection operations must be performed between the two sets when executing operations such as a spatial join if no preprocessing sorting step has been applied (e.g., [11, 49, 67]), although a number of good algorithms have been devised for spatial joins for object-based representations (e.g., [52, 53]). In contrast, disjoint spatial indexing methods that make use of a regular decomposition of the underlying space such as the region quadtree and bintree (a regular decomposition k-d tree) are good when operating on two different data sets as the occupied space in the two sets is correlated thereby simplifying the spatial join algorithms which makes them preferable to disjoint spatial indexing methods that do not employ regular decomposition such as the R^+-tree (e.g., [42]). Nevertheless, there is the cost of dealing with duplicate answers (as mentioned above) which is incurred regardless of which disjoint method is used. Thus there is no one best or optimal representation. Ultimately, users make their decision on the basis of what is important to them, possibly making use of cost models (e.g., [8, 78]).

There are a number of directions for future research. One direction is the further investigation of the question of the construction of an R-tree where overflowing nodes are split so that the regions that correspond to the two resulting children are either disjoint or the region corresponding to one child is completely contained in the region corresponding to the other child. Another interesting direction is the investigation of the use of decoupling the partitioning and grouping processes in representations of other types of data besides point data. A preliminary study has been conducted [21] where the PK-tree was used to store spatial data with extent such as collections of line segments. In this case, the data has extent and instead of mapping each line segment into a point in a space of higher dimensions and then using a PK PR quadtree (as suggested and used by Wang et al. [79]), the line segments were represented using a spatial index such as the PMR quadtree [57, 58] and the resulting blocks stored in a PK-tree, termed a PK PMR quadtree [21]. Performance comparisons with an R*-tree [13] showed that the PK PMR quadtree resulted in good query performance while being much faster to construct [21]. A related decoupling approach is used by De Floriani et al. [24] in the representations of triangular decompositions of surfaces such as terrain data. The interrelationship of the principles used in this representation with the PK-tree and BV-tree is worthy of investigation. In particular, it may be attractive to impose a spatial index on this representation and store it in a PK-tree.

It is interesting to point out that the PK-tree, although quite different from the BV-tree, has some similarities to the hB-tree and BANG file, which are ancestors of the BV-tree. When the PK-tree is built using the PR k-d tree partitioning process (i.e., a PK PR k-d tree), the result resembles the BD-tree [60, 61], which is also a bucket method, but the PK-tree has the advantage that it is uniquely defined while the BD-tree is not. However, the non-uniqueness of the BD-tree does enable the BD-tree to be varied so that a more balanced structure can be achieved. The price of the balance is that the BD-tree does not decompose the underlying space into squares. Instead, the decomposition is into a collection of 'holey bricks' (i.e., squares that possibly have smaller squares cut out) similar to that resulting from use of the hB-tree and the BANG file. This makes subsequent processing a bit difficult. Also, all but the leaf nodes in the BD-tree have a fanout of two whereas the fanout of all of the PK PR k-d tree nodes is much higher. The limited fanout of the nonleaf nodes in the BD-tree reduces the motivation for using it as a disk-resident data structure.

The PK-tree is also related to the k-d-B-trie (e.g., multilevel grid file [50], BANG file [32], and buddy tree [75]) which is a regular decomposition variant of the k-d-B-tree. The similarity is that the bucketing condition in the k-d-B-trie is only in terms of the nonempty regions, while in the PK-tree, the *k*-instantiation is also only in terms of these nonempty regions. However, there is an important difference which is that in the PK-tree no distinction is made between data and directory nodes — that is, the PK-tree consists of the *k*-instantiated nodes of the corresponding partition tree (which may be data or directory nodes). Thus points (or other objects) can appear in both leaf and nonleaf nodes of the PK-tree. This has the ramification that groups of points can stay grouped together in the same node for a much longer time period than if bucketing was used. In particular, when using a bucket method, once a pair of points are placed in the same bucket, they remain in the same bucket until it overflows at which time they may be separated, whereas in the PK-tree these points are often moved between nodes as dictated by *k*-instantiation considerations. A drawback of having the points in both leaf and nonleaf nodes is that the fanout of the nonleaf nodes is potentially reduced thereby leading to longer search paths since the height of the tree is longer. On the other hand, a drawback of storing only points in leaf nodes, as in the k-d-B-trie, is that the storage utilization of the leaf nodes may be low.

Acknowledgements

I have benefitted greatly from discussions with Houman Alborzi and Gisli R. Hjaltason. I thank Gisli R. Hjaltason for drawing the figures and Edwin Jacox for help in the final preparation for publication. I would also like to thank the anonymous reviewers for their detailed comments which greatly sharpened the presentation of the ideas in this paper.

References

[1] Compatibility of wireless services with enhanced 911. *Federal Register*, pages 40348–40352, August 1996.

[2] Wireless radio services; compatibility with enhanced 911 emergency calling systems. *Federal Register*, pages 60126–60131, November 1999.

[3] D. J. Abel. A B^+–tree structure for large quadtrees. *Computer Vision, Graphics, and Image Processing*, 27(1):19–31, July 1984.

[4] C. Aggarwal, J. Wolf, P. Yu, and M. Epelman. The *S*-tree: an efficient index for multidimensional objects. In *Advances in Spatial Databases — 5th International Symposium, SSD'97*, M. Scholl and A. Voisard, eds., pages 350–373, Berlin, Germany, July 1997. Also Springer-Verlag Lecture Notes in Computer Science 1262.

[5] H. Alborzi and H. Samet. Bv-trees and r-trees for collections of extended objects represented by their minimum bounding hyper-rectangles, 2004. Not published.

[6] A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, July 1993.

[7] W. G. Aref and H. Samet. Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 178–189, Charleston, SC, August 1992.

[8] W. G. Aref and H. Samet. A cost model for query optimization using r-trees. In *Proceedings of the 2nd ACM Workshop on Geographic Information Systems*, N. Pissinou and K. Makki, eds., pages 60–67, Gaithersburg, MD, December 1994.

[9] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM)*, pages 347–354, Gaithersburg, MD, December 1994.

[10] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation (ALENEX'99)*, M. T. Goodrich and C. C. McGeoch, eds., pages 328–348, Baltimore, MD, January 1999. Also Springer-Verlag Lecture Notes in Computer Science 1619.

[11] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, A. Gupta, O. Shmueli, and J. Widom, eds., pages 570–581, New York, August 1998.

[12] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[13] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[14] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[15] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *Advances in Database Technology — EDBT'98, Proceedings of the 6th International Conference on Extending Database Technology*, pages 216–230, Valencia, Spain, March 1998.

[16] S. Berchtold, C. Böhm, H.-P. Kriegel, J. Sander, and H. V. Jagadish. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 577–588, San Diego, CA, February 2000.

[17] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds., pages 28–39, Mumbai (Bombay), India, September 1996.

[18] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, eds., pages 406–415, Athens, Greece, August 1997.

[19] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is “nearest neighbor” meaningful? In *Proceedings of the 7th International Conference on Database Theory (ICDT’99)*, C. Beeri and P. Buneman, eds., pages 217–235, Berlin, Germany, January 1999. Also Springer-Verlag Lecture Notes in Computer Science 1540.

[20] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[21] F. Brabec, G. R. Hjaltason, and H. Samet. Indexing spatial objects with the PK-tree. Unpublished Manuscript, 2001.

[22] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proceedings of the ACM SIGMOD Conference*, pages 197–208, Minneapolis, MN, June 1994.

[23] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[24] L. De Floriani, P. Marzano, and E. Puppo. Multiresolution models for topographic surface description. *The Visual Computer*, 12(7):317–345, August 1996.

[25] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 535–546, San Diego, CA, February 2000.

[26] G. Evangelidis, D. Lomet, and B. Salzberg. Node deletion in the hB$^{\Pi}$-tree. Technical Report NU-CCS-94-04, Northeastern University, Boston, 1994.

[27] G. Evangelidis, D. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: A modified hb-tree supporting concurrency, recovery and node consolidation. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, U. Dayal, P. M. D. Gray, and S. Nishio, eds., Zurich, Switzerland, September 1995.

[28] G. Evangelidis, D. Lomet, and B. Salzberg. The hB$^{\Pi}$-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB Journal*, 6(1):1–25, 1997.

[29] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 247–252, Philadelphia, March 1989.

[30] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proceedings of the ACM SIGMOD Conference*, pages 426–439, San Francisco, May 1987.

[31] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *Proceedings of the 9th International Conference on Information and Knowledge Management (CIKM)*, pages 202–209, McLean, VA, November 2000.

[32] M. Freeston. The BANG file: a new kind of grid file. In *Proceedings of the ACM SIGMOD Conference*, pages 260–269, San Francisco, May 1987.

[33] M. Freeston. A general solution of the n-dimensional B-tree problem. Computer Science Department ECRC-94-40, ECRC, Munich, Germany, 1994.

[34] M. Freeston. A general solution of the n-dimensional B-tree problem. In *Proceedings of the ACM SIGMOD Conference*, pages 80–91, San Jose, CA, May 1995.

[35] M. Freeston. On the complexity of bv-tree updates. In *Constraint Databases and Their Applications — 2nd International Workshop on Constraint Database Systems,CDB'97*, V. Gaede, A. Brodsky, O. Günther, D. Srivastava, V. Vianu, and M. Wallace, eds., pages 282–293, Delphi Greece, January 1997. Also Springer-Verlag Lecture Notes in Computer Science 1191.

[36] M. W. Freeston. Advances in the design of the BANG file. In *Proceedings of the 3rd International Conference on Foundations of Data Organization and Algorithms (FODO)*, W. Litwin and H.-J. Schek, eds., pages 322–338, Paris, France, June 1989. Also Springer-Verlag Lecture Notes in Computer Science 367.

[37] V. Gaede. Optimal redundancy in spatial database systems. In *Advances in Spatial Databases — 4th International Symposium, SSD'95*, M. J. Egenhofer and J. R. Herring, eds., pages 96–116, Portland, ME, August 1995. Also Springer-Verlag Lecture Notes in Computer Science 951.

[38] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the SIGGRAPH'96 Conference*, pages 171–180, New Orleans, August 1996.

[39] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, June 1984.

[40] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional point and non-point data. In *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, P. M. G. Apers and G. Wiederhold, eds., pages 45–53, Amsterdam, The Netherlands, August 1989.

[41] D. Hilbert. Ueber stetige abbildung einer linie auf flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[42] E. G. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, U. Dayal, P. M. D. Gray, and S. Nishio, eds., pages 606–618, Zurich, Switzerland, September 1995.

[43] P. M. Hubbard. Collision detection for interactive computer graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, September 1995.

[44] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the ACM SIGMOD Conference*, pages 332–342, Atlantic City, NJ, June 1990.

[45] H. V. Jagadish. Spatial search with polyhedra. In *Proceedings of the 6th IEEE International Conference on Data Engineering*, pages 311–319, Los Angeles, February 1990.

[46] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference*, J. Peckham, ed., pages 369–380, Tucson, AZ, May 1997.

[47] A. Klinger. Patterns and search statistics. In *Optimizing Methods in Statistics*, J. S. Rustagi, ed., pages 303–337. Academic Press, New York, 1971.

[48] G. D. Knott. Expandable open addressing hash table storage and retrieval. In *Proceedings of SIGFIDET Workshop on Data Description, Acess and Control*, pages 187–206, San Diego, CA, November 1971.

[49] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proceedings of the ACM SIGMOD Conference*, J. Peckham, ed., pages 324–335, Tucson, AZ, May 1997.

[50] R. Krishnamurthy and K.-Y. Whang. Multilevel grid files. Technical report, IBM T. J. Watson Research Center Technical Report, Yorktown Heights, NY, 1985.

[51] J. Li, D. Rotem, and J. Srivastava. Algorithms for loading parallel grid files. In *Proceedings of the ACM SIGMOD Conference*, pages 347–356, Washington, DC, May 1993.

[52] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the ACM SIGMOD Conference*, pages 209–220, Minneapolis, MN, June 1994.

[53] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proceedings of the ACM SIGMOD Conference*, pages 247–258, Montréal, Canada, June 1996.

[54] D. Lomet and B. Salzberg. A robust multi-attribute search structure. In *Proceedings of the 5th IEEE International Conference on Data Engineering*, pages 296–304, Los Angeles, February 1989.

[55] D. Lomet and B. Salzberg. The hB–tree: a multi-attribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990. Also Northeastern University Technical Report NU-CCS-87-24.

[56] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. , IBM Ltd., Ottawa, Canada, 1966.

[57] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, August 1986. Also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, TX, August 1986.

[58] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proceedings of the ACM SIGMOD Conference*, pages 270–277, San Francisco, May 1987.

[59] S. Nilsson and M. Tikkanen. An experimental study of compression methods for dynamic tries. *Algorithmica*, 33(1):19–33, 2002.

[60] Y. Ohsawa and M. Sakauchi. The BD-Tree — a new n-dimensional data structure with highly efficient dynamic characteristics. In *Information Processing'83*, R. E. A. Mason, ed., pages 539–544, North-Holland, Paris, France, September 1983.

[61] Y. Ohsawa and M. Sakauchi. Multidimensional data management structure with efficient dynamic characteristics. *Systems, Computers, Controls*, 14(5):77–87, 1983. (translated from, *Denshi Tsushin Gakkai Ronbunshi 66–D*, 10(October 1983), 1193–1200).

[62] S. M. Omohundro. Five balltree construction algorithms. Technical Report TR-89-063, International Computer Science Institute, Berkeley, CA, December 1989.

[63] P. van Oosterom and E. Claassen. Orientation insensitive indexing methods for geometric objects. In *Proceedings of the 4th International Symposium on Spatial Data Handling,* vol. 2, pages 1016–1029, Zurich, Switzerland, July 1990.

[64] J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982.

[65] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS),* pages 181–190, Waterloo, Ontario, Canada, April 1984.

[66] M. A. Ouksel and O. Mayer. A robust and efficient spatial data structure. *Acta Informatica,* 29(4):335–373, July 1992.

[67] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD Conference*, pages 259–270, Montréal, Canada, June 1996.

[68] D. R. Reddy and S. Rubin. Representation of three–dimensional objects. Computer Science Department CMU–CS–78–113, Carnegie–Mellon University, Pittsburgh, April 1978.

[69] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 10–18, Ann Arbor, MI, April 1981.

[70] A. Rosenfeld, H. Samet, C. Shaffer, and R. E. Webber. Application of hierarchical data structures to geographical information systems: phase II. Computer Science Department TR-1327, University of Maryland, College Park, MD, September 1983.

[71] B. Salzberg. *File structures: An analytic approach.* Prentice–Hall, Englewood Cliffs, NJ, 1988.

[72] B. Salzberg. On indexing spatial and temporal data. *Information Systems*, 19(6):447–465, September 1994.

[73] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

[74] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[75] B. Seeger and H.-P. Kriegel. The buddy-tree: an efficient and robust access method for spatial data base systems. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB)*, D. McLeod, R. Sacks-Davis, and H.-J. Schek, eds., pages 590–601, Brisbane, Australia, August 1990.

[76] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+–tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB),* P. M. Stocker and W. Kent, eds., pages 71–79, Brighton, United Kingdom, September 1987. Also Computer Science TR-1795, University of Maryland, College Park, MD.

[77] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the 1st International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986.

[78] Y. Theodoridis, E. Stefanakis, and T. K. Sellis. Efficient cost models for spatial queries using R-trees. *IEEE Transactions on Knowledge and Data Engineering*, 12(1):19–32, January/February 2000.

[79] W. Wang, J. Yang, and R. Muntz. PK-tree: a spatial index structure for high dimensional point data. In *Proceedings of the 5th International Conference on Foundations of Data Organization and Algorithms (FODO)*, K. Tanaka and S. Ghandeharizadeh, eds., pages 27–36, Kobe, Japan, November 1998.

[80] W. Wang, J. Yang, and R. Muntz. PK-tree: a spatial index structure for high dimensional point data. Computer Science Department Report 980032, University of California, Los Angeles, September 1998.

[81] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, A. Gupta, O. Shmueli, and J. Widom, eds., pages 194–205, New York, August 1998.

[82] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th IEEE International Conference on Data Engineering*, S. Y. W. Su, ed., pages 516–523, New Orleans, February 1996.

[83] J. Yang, W. Wang, and R. Muntz. Yet another spatial indexing structure. Computer Science Department Technical Report 97040, University of California at Los Angeles (UCLA), Los Angeles, 1997. (see `http://dml.cs.ucla.edu/~weiwang/paper/TR97040.ps`).