The 2DR-tree: A 2-dimensional spatial access method*

Wendy Osborn and Ken Barker
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, AB
T2N 1N4 Canada
email:{osborn,barker}@cpsc.ucalgary.ca

Abstract

This paper presents the 2DR-tree, a novel approach for accessing spatial data. The 2DR-tree uses nodes that are the same dimensionality as the data space. Therefore, all relationships between objects are preserved and different searching strategies such as binary and greedy are supported. The insertion and deletion strategies both use a binary partition of a node to insert an object or update a non-leaf minimum bounding rectangle. A validity test ensures that each node involved in an insertion or deletion preserves the spatial relationships among its objects. A performance evaluation shows the advantages of the 2DR-tree and identifies issues for future consideration.

1 Introduction

Many applications rely on spatial data. Some include geographical information systems (GIS) and bioinformatics databases. For example, Alberta En-

*University of Calgary Technical Report 2004-750-15. This research funded by an NSERC Discovery Grant.

ergy and Utilities Board [AEUB04] maintains a GIS on the provincial geological survey for energy and mineral resources. The Research Collaboratory for Structural Bioinformatics [Bio04] maintains a repository of three-dimensional protein structures. These "non-standard" databases contain large amounts of data that is represented in multidimensional space. An important issue for large spatial data sets is efficient retrieval of specific objects or subsets of spatial objects.

No n-dimensional to one-dimensional mapping of spatial data that preserves all spatial relationships exists [GG98]. However, most multidimensional indexing structures proposed in the literature extend hierarchical data structures originally proposed for one-dimensional data. These structures lead to several problems including: unnecessary searching, unnecessary entry checking in nodes, and superlinear growth. Attempts to map n-dimensional data to a one-dimensional sequence of values do not maintain all spatial relationships between objects.

A new hierarchical spatial access method, the nDR-tree, is proposed that fits the existing data by using nodes that are the same dimensionality. The two-dimensional version, the 2DR-tree, is presented here which uses two-dimensional nodes to index two-dimensional data. The minimum bounding rectangles in each node are organized according to a "validity rule" that preserves spatial relationships. This provides support for both binary searching that takes advantage of spatial relationships and greedy searching that reduces the number of minimum bounding rectangles within a node that must be tested.

The remainder of the paper is organized as follows. Section 2 presents related work in the area of spatial access methods. Section 3 introduces the 2DR-tree and defines key concepts. Section 3.1 presents the node validity strategy used by the tree. Section 4 presents the insertion strategy and Section 5 presents the deletion strategy. Section 6 shows insertion and deletion examples. Section 7 presents performance evaluations for the insertion and deletion algorithms. Finally, Section 8 presents conclusions and future research directions.

2 Related Work

Many approaches for indexing spatial data are proposed in the literature. In addition, several surveys [GG98, Sam90, RSV01] have been published. Spatial access methods are classified into three categories [GG98]: main memory access methods [FB74, Ben75, Ben79], point access methods [FBF77, Ben79, Rob81, HSW89, NHS81, NHS84, Hin85] and spatial access methods [Gut84, BKSS90, BKK96, AWYE97, CPZ97, KF94, SRF87, OM84, Ore89, Bay97, SK96, Kou00].

Main memory access methods extend the Binary Search Tree (BST) to handle sets of attribute values (i.e. points). These methods recursively partition the space based on different attribute values. The quadtree [FB74] recursively partitions the space into quadrants using the dimension values stored in each node. The k-d tree [Ben75, Ben79] performs an alternating recursive binary partition using the assigned dimension value stored in each node. Both structures are easy to implement. However, both have limitations including: point data, skewing in the presence of dynamic data, and lack of secondary storage.

Point access methods (PAMs) are space-partitioning techniques that take secondary storage and, to some extent, height balancing into consideration. PAMs are classified into hierarchical and non-hierarchical methods. Hierarchical PAMs recursively partition the space using a tree structure, but are organized for efficient access from secondary storage. Hierarchical PAMs include the adaptive k-d tree [FBF77, Ben79] and the k-d-B tree [Rob81] which also takes balancing into consideration. Hierarchical (i.e., grid file) PAMS [NHS81, NHS84] extends the directory structure from extendible hashing to handle point data. Unfortunately, grid files experience superlinear growth and many empty buckets.

Spatial access methods (SAMs) provide uniform access to both point and object data. In addition, they remain height-balanced when dynamic data is present. SAMs are classified [GG98] into approximation methods, clipping methods, and mapping methods. Approximation methods store a hierarchy of minimum bounding rectangles that represent both the objects and the space they occupy. Since the space is not partitioned, minimum bounding rectangles can overlap. Many approximation SAMs are proposed, including the R-tree [Gut84], the R^*-tree [BKSS90], X-tree [BKK96], the M-tree [CPZ97], and the S-tree [AWYE97]. Clipping methods, such as the R^+-tree [SRF87], partition an object to be stored in non-overlapping minimum bounding rectangles. Mapping methods provide a one-dimensional ordering for n-dimensional objects, which allows storage and access using an existing one-dimensional access method such as a B-tree [BM74]. Approaches using mapping include Z-ordering [OM84, Ore89], the Hilbert R-tree [KF94], the

Universal B-tree [Bay97], and the Filter tree [SK96, Kou00].

Günther and Gaede [GG98] state that no n-dimensional to one-dimensional mapping of spatial data exists that preserves all spatial relationships. A limitation to most proposed access methods is their one-dimensional structure that requires forcing n-dimensional data to fit them. This one-dimensional limitation leads to inefficient searching, both within the structure and within a node itself, since the only option is a linear search. Grid files are n-dimensional in structure, but have superlinear growth and possibly many empty buckets. Mapping methods do not preserve all spatial relationships.

3 The 2DR-tree

The 2DR-tree is a hierarchical spatial data structure where each node is two-dimensional. A minimum bounding rectangle is stored in an appropriate location with respect to the spatial location of all other objects in the node. Using two-dimensional nodes allows minimum bounding rectangles to be placed so that the spatial relationships are preserved.

A minimum bounding rectangle defines the minimum extent along both the x-axis and the y-axis that encompass either an object or a collection of minimum bounding rectangles. The extent along an axis is defined using lower and upper co-ordinate values of an object or a collection of objects. The centroid of a minimum bounding rectangle is the co-ordinate pair (i, j) of its centre. The coverage of a minimum bounding rectangle is the total area covered by the rectangle. The coverage of a tree is the total coverage of all minimum bounding rectangles in the tree. The overcoverage of a minimum bounding rectangle is the extra space in the rectangle not covered by other minimum bounding rectangles it contains. The overcoverage of a tree is the total overcoverage of all minimum bounding rectangles in the tree. In addition, overlap exists because the minimum bounding rectangles are not disjoint. The overlap of a tree is the total overlap between all pairs of minimum bounding rectangles. Although overlap-free coverage of minimum bounding rectangles is desirable, it has been proven in [RL85] that zero overlap cannot be guaranteed for any set of objects.

For each node N, X is the number of index locations along the x-axis, and Y is the number of index locations along the y-axis. The order of N is $O = X * Y$. All nodes in a 2DR-tree have the same order. Therefore, the order of a 2DR-tree is the order of its nodes. $N_{(i,j)}$ is location (i, j) in node

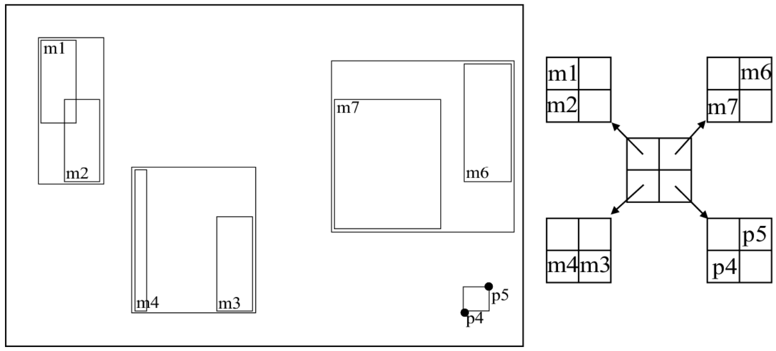


Figure 1: Order 4*4 2DR-tree

N. Each location stores:

$$(MBR_{(i,j)}, ptr_{(i,j)})$$

where $MBR_{(i,j)}$ is a minimum bounding rectangle and $ptr_{(i,j)}$ is a pointer. In a leaf node, $MBR_{(i,j)}$ encloses an object and $ptr_{(i,j)}$ references the object on secondary storage. In a non-leaf node, $MBR_{(i,j)}$ encloses all minimum bounding rectangles in the subtree referenced by $ptr_{(i,j)}$.

A node region (lx, hx, ly, hy) is a two-dimensional subset of (x, y) index locations in a node. The index values lx and hx are the lower and upper bounds of the node region along the x-axis. The index values ly and hy are the lower and upper bounds of the node region along the y-axis.

A node object set is the set of objects in a node. A node space is the region occupied by a node objects set. This is equal to the area of the minimum bounding rectangle that encloses a node object set.

Figure 1 shows an order 2*2 2DR-tree that preserves all spatial relationships for the given data set. Beginning with the leaf node descending from root location $(0, 0)$, the centroid for m3 is located southeast of the centroid so m3 is located east of m4 in the node (m4,m3). In node (m5,p5,m8), p5 is located southeast of the centroid for m5, and the centroid for m8 is located northeast of the centroid for m5 and northwest of p5. Therefore, p5 is stored east of m5 while m8 is stored northeast of m5 and north of p5. The remaining two leaf nodes, (m7,m9) and (m2,m1,p9) preserve the spatial relationships between the objects each contains. Spatial relationships between minimum bounding rectangles in the root node are also preserved.

3.1 Node Validity

Both insertion and deletion require searching. To employ different searching strategies, the spatial relationships between minimum bounding rectangles in each node must be preserved. For each location $N_{(i,j)}, i = 0 \ldots (X-1), j = 0 \ldots (Y-1)$ in node N, if $N_{(i,j)}$ contains a minimum bounding rectangle $MBR_{(i,j)}$,

- Location $N_{(k,l)}, k = (i+1) \ldots (X-1), l = 0 \ldots j$ contains $MBR_{(k,l)}$ whose centroid is southeast of the centroid for $MBR_{(i,j)}$,
- Location $N_{(k,l)}, k = (i+1) \ldots (X-1), l = (j+1) \ldots (Y-1)$ contains $MBR_{(k,l)}$ whose centroid is northeast of the centroid for $MBR_{(i,j)}$, and
- Location $N_{(k,l)}, k = 0 \ldots i, l = (j+1) \ldots (Y-1)$ contains $MBR_{(k,l)}$ whose centroid is northwest of the centroid for $MBR_{(i,j)}$.

A validity testing strategy ensures that each minimum bounding rectangle is in a proper location with respect to all other minimum bounding rectangles in the node. Each minimum bounding rectangle at location $(i,j), i = 0 \ldots (X-1), j = 0 \ldots (Y-1)$ is compared with every other minimum bounding rectangle in the following way. First, the spatial relationship between (i,j) and all $(k,l), k = 0 \ldots i, l = (j+1) \ldots (Y-1)$ are compared to see if all (k,l) are northwest of (i,j). Second, the spatial relationship between (i,j) and all $(k,l), k = (i+1) \ldots (X-1), l = (j+1) \ldots (Y-1)$ are compared to see if all (k,l) are northeast of (i,j). Finally, the spatial relationship between (i,j) and all $(k,l), k = (i+1) \ldots (Y-1), l = 0 \ldots j$ are compared to see if all (k,l) are southeast of (i,j). The validity test starts at location $(0,0)$ and continues through all locations until all minimum bounding rectangles are validated, or an invalidly located minimum bounding rectangle is identified.

4 Insertion

This section describes the 2DR-tree insertion strategy, which exploits the maintained spatial relationships. The strategy performs a greedy search through the non-leaf nodes to find an appropriate leaf node to insert new minimum bounding rectangle MBR^n. It then performs a node reduction to find an appropriate location in the node for MBR^n. After inserting the minimum bounding rectangle, the leaf node is restored and all minimum bounding rectangles along the insertion path are updated. Overflow and node invalidity are handled using different splitting strategies. The insertion strategy, specifically the greedy search, node reduction, node restoration, insertion path update, and splitting strategies are discussed below.

4.1 Leaf Node Search

The leaf node search strategy finds the appropriate node into which to store the new object. The insertion path contains one non-leaf node from each level. Each non-leaf node contains the minimum bounding rectangle that requires either a minimal or the least area increase necessary to include the new object. The appropriate minimum bounding rectangle is found using either a greedy search or a complete search of each node.

The greedy search finds a minimum bounding rectangle requiring a small area increase to include MBR^n. The search finds this minimum bounding rectangle along the path of decreasing area increases, and stops when the area increase begins to grow. The search may not find the minimum bounding rectangle requiring the least area increase but the number of minimum bounding rectangles involved in the search is reduced.

Each minimum bounding rectangle on the search path (i, j) is compared with $(i, j+1)$,$(i+1, j+1)$ and $(i+1, j)$. If the minimum bounding rectangle at (i, j) has the smallest area increase, the search terminates in the node and continues in the corresponding subtree. Otherwise, the search continues in the direction that contains a minimum bounding rectangle with a smaller area increase. Any ties in the minimum area increase between minimum bounding rectangles are broken by choosing the one with the smallest area. The greedy search begins at location $(0, 0)$ in the root and continues until the minimum area increase along the search path stops decreasing or the node boundaries are reached. The search continues in all subtrees corresponding to the chosen minimum bounding rectangles until a leaf node is reached.

The greedy search is attempted first when locating an appropriate minimum bounding rectangle. The search requires a "seed" minimum bounding rectangle, which must exist either in $(0, 0)$ (ideal), or one of $(0, 1)$, $(1, 1)$, or $(1, 0)$. However, in a few instances, a "seed" minimum bounding rectangle may not exist. This occurs during updating when parent minimum bounding rectangles are removed and re-inserted elsewhere. Figure 2 shows an example node that cannot be searched using the greedy search strategy. The locations $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$ do not contain the minimum bounding rectangle needed to seed the greedy search. The complete search scans

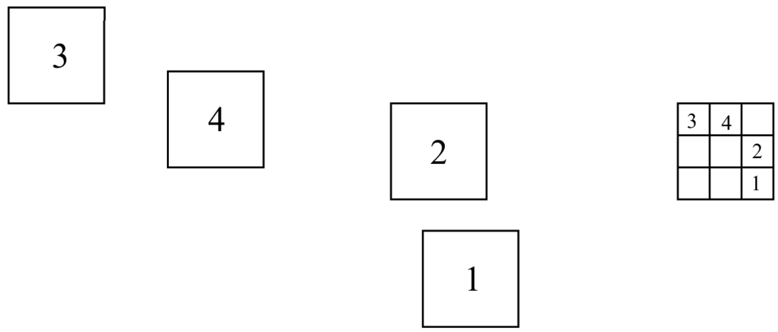


Figure 2: Greedy Search Failure

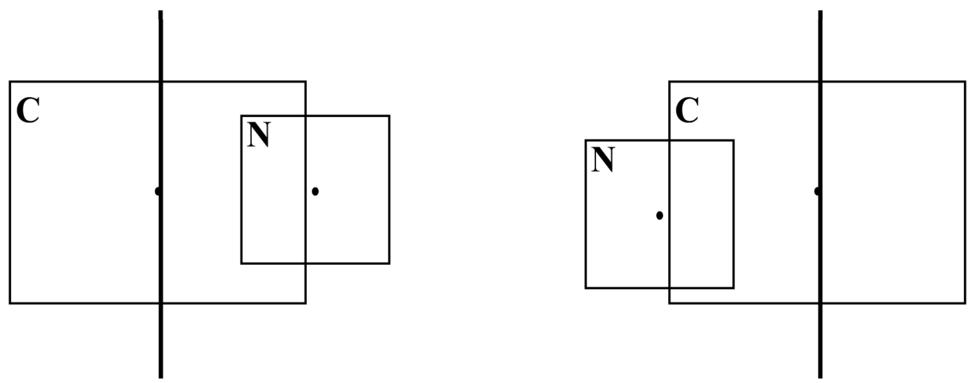
all minimum bounding rectangles to find one with the least area increase required to include the new object.

4.2 Node Reduction

After a leaf node is found to insert MBR^n, a location is found in the node for the minimum bounding rectangle. The node reduction strategy performs a recursive binary partition of the node space to find this location. During each iteration, the node space is partitioned through its centroid along the dimension with the longest extent. Second, each subspace is mapped to a node region. Third, one subspace is chosen for the next iteration, while the node region corresponding to the other subspace is removed from the node and saved for restoration after MBR^n is inserted.

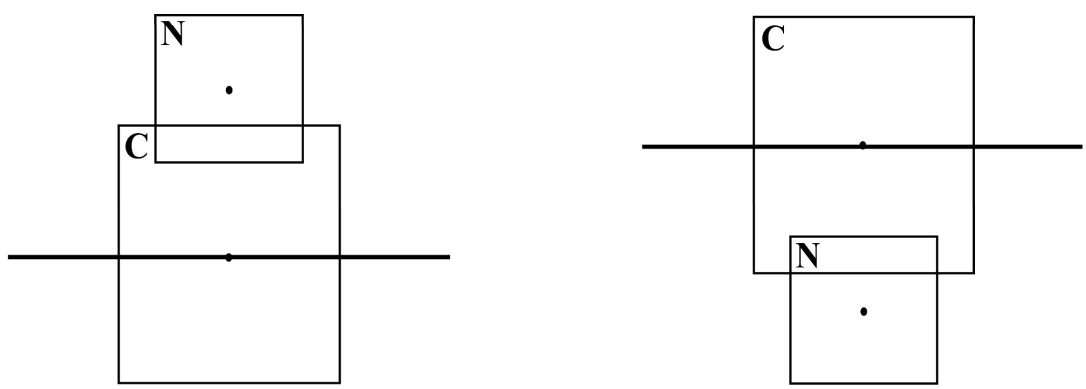
Figure 3 shows an X-partition of the node space. In Figure 3.a, the new object N is located east of the partition so the node region that corresponds to the east subspace is chosen for further partitioning. In Figure 3.b, N is located west of the partition so the node region corresponding to the west subspace is chosen for further partitioning. Figure 4 shows a Y-partition of the node space. Similarly, the subspace that contains N is chosen for further partitioning.

The mapping of each subspace to a node region is performed in the following manner. First, the split point sp corresponding to the node space partition is found for the original node region $(lx_{ons}, hx_{ons}, ly_{ons}, hy_{ons})$. The original node region contains all objects in the node. This split point is found using one of the following strategies.



a. New East of Partition b. New West of Partition

Figure 3: Node Reduction - X-Split Reduction Cases



a. New North of Partition b. New South of Partition

Figure 4: Node Reduction - Y-Split Reduction Cases

1. X-split. The search for a split point begins at the west end of the node region $(lx_{ons}, hx_{ons}, ly_{ons}, hy_{ons})$. Proceeding from west to east, each column is scanned to find one minimum bounding rectangle whose centroid is west of the centroid for the node space. If the column contains one such minimum bounding rectangle, the split point is moved one column to the east. This continues until a column is reached that contains no minimum bounding rectangles with centroids west of the centroid for the node space; or until the eastern most column is checked.

2. Y-split. The search for a split point begins at the south end of the node region $(lx_{ons}, hx_{ons}, ly_{ons}, hy_{ons})$. Proceeding from south to north, each row is scanned to find one minimum bounding rectangle whose centroid is south of the centroid for the node space. If the row contains one such minimum bounding rectangle, the split point is moved one row to the north. This continues until a row is reached that contains no minimum bounding rectangles with centroids south of the centroid for the node space; or until the northern most row is checked.

Next, one subspace becomes the reduced node space and is mapped to a reduced node region $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns})$. The other subspace is mapped to node region (lx_d, hx_d, ly_d, hy_d) where d is n (north), s (south), e (east) or w (west). If the new object is east of an X-split, then the subspace that is east of the partition becomes the reduced node space and the corresponding reduced node region $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns})$ is equal to $(sp + 1, hx_{ons}, ly_{ons}, hy_{ons})$. The subspace that is west of the partition becomes the node space that is saved on the stack and the corresponding node region (lx_w, hx_w, ly_w, hy_w) is equal to $(lx_{ons}, sp, ly_{ons}, hy_{ons})$. Similarly, node spaces are mapped when the new object is west, north or south of a node space partition.

Node reduction now continues using the chosen subspace until either one minimum bounding rectangle remains in the node (MBR^r), or multiple objects exist and MBR^n has some specific spatial relationship with those objects. Figure 5 shows the base cases involving one object. Figure 6 shows the other canonical cases involving multiple objects.

The base cases involving one minimum bounding rectangle are discussed first:

1. North Insert. In Figure 5.a, the centroid for MBR^n is located northwest of the centroid for MBR^r and the slope of a line joining the centroids

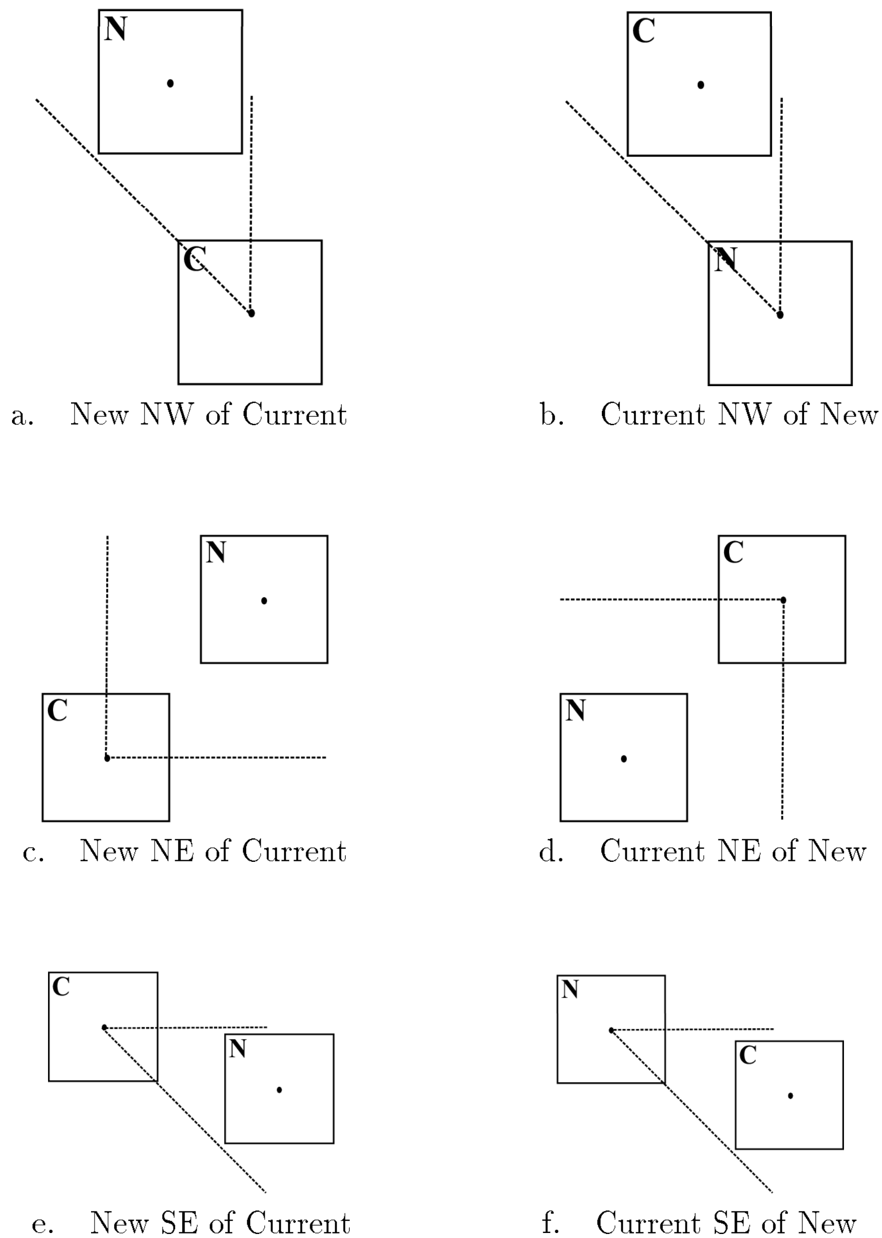


Figure 5: Node Reduction - Single-Object Base Cases

is less than -1. Given location (i, j) for MBR^r, MBR^n is inserted north in location $(i, j{+}1)$. The updated node region is $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns} + 1)$.

2. North Swap Insert. In Figure 5b, the centroid for MBR^n is located southeast of the centroid for MBR^r and the slope of a line joining the centroids is less than -1. Given location (i, j) for MBR^r, MBR^n is swapped with MBR^r and MBR^r is inserted north of MBR^n in location $(i, j + 1)$. The updated node region is $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns} + 1)$.

3. East Insert. In Figure 5c, the centroid for MBR^n is located southeast of the centroid for MBR^r and the slope of a line joining the centroids is greater than -1. Given location (i, j) for MBR^r, MBR^n is inserted east of MBR^r in location $(i + 1, j)$. The updated node region is $(lx_{rns}, hx_{rns} + 1, ly_{rns}, hy_{rns})$.

4. East Swap Insert. In Figure 5c, the centroid for MBR^n is located northwest of the centroid for MBR^r and the slope of a line joining the centroids is greater than -1. Given location (i, j) for MBR^r, MBR^n is swapped with MBR^r and MBR^r is inserted east of MBR^n in location $(i + 1, j)$. The updated node region is $(lx_{rns}, hx_{rns} + 1, ly_{rns}, hy_{rns})$.

5. Northeast Insert. In Figure 5e the centroid for MBR^n is located northeast of the centroid for MBR^r. Given location (i, j) for MBR^r, MBR^n is inserted northeast of MBR^r in location $(i + 1, j + 1)$. The updated node region is $(lx_{rns}, hx_{rns} + 1, ly_{rns}, hy_{rns} + 1)$.

6. Northeast Swap Insert. In Figure 5.f, the centroid for MBR^r is located northeast of the centroid for MBR^n. Given location (i, j) for MBR^r, MBR^n is swapped with MBR^r and MBR^r is inserted northeast of MBR^n in location $(i + 1, j + 1)$. The updated node region is $(lx_{rns}, hx_{rns} + 1, ly_{rns}, hy_{rns} + 1)$.

The multiple object base cases identify situations where MBR^n has certain spatial relationships with respect to all objects in a node region. These attempt to reduce the number of node regions generated during node reduction because restoring each requires a costly validity test. The multi-object cases are discussed next:

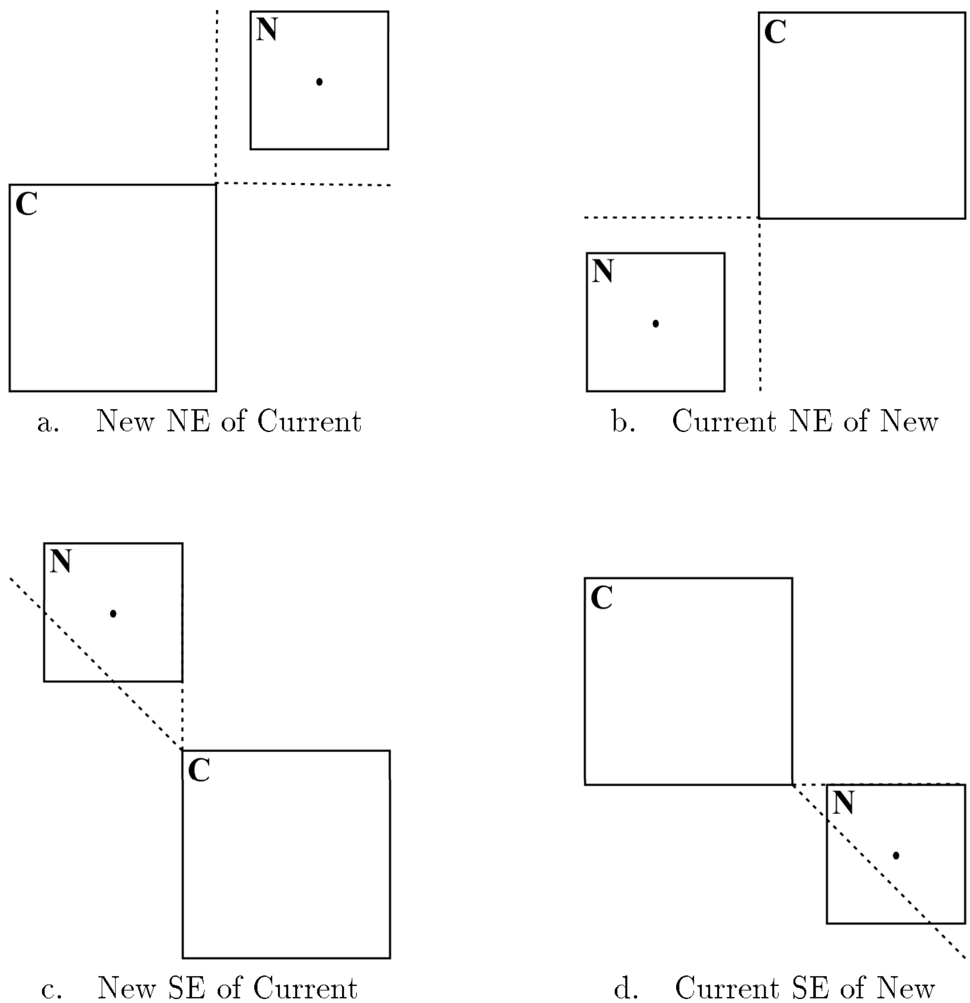


Figure 6: Node Reduction - Multiple-Object Base Cases

1. North Insert. In Figure 6.a, the centroid for MBR^n is located northwest of all remaining minimum bounding rectangles in the node space and the slope of a line between the centroids of the minimum bounding rectangle and the node space is less than -1. Given node region $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns})$ for the node space, MBR^n is inserted in location $(lx_{rns}, hy_{rns} + 1)$. The updated node region is $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns} + 1)$.

2. East Insert. In Figure 6.b, the centroid for MBR^n is located southeast of all remaining minimum bounding rectangles in the node space and the slope of a line between the centroids of the minimum bounding rectangle and the node space is greater than -1. Given node region $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns})$ for the node space, MBR^n is inserted in location $(hx_{rns} + 1, ly_{rns})$. The updated node region is $(lx_{rns}, hx_{rns} + 1, ly_{rns}, hy_{rns})$.

3. Northeast Insert. In Figure 6.c, the centroid for MBR^n is located northeast of all remaining minimum bounding rectangles in the node space. Given node region $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns})$ for the node space, MBR^n is inserted in location $(hx_{rns} + 1, hy_{rns} + 1)$. The updated node region is $(lx_{rns}, hx_{rns} + 1, ly_{rns}, hy_{rns} + 1)$.

4. Northeast Swap Insert. In Figure 6.d, the centroid for MBR^n is located southwest of all remaining minimum bounding rectangles in the node space. Given node region $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns})$ for the node space, the node region is shifted to the northeast (i.e. over one column and up one row), and MBR^n is inserted into the original (lx_{rns}, ly_{rns}) location. The updated node region is $(lx_{rns}, hx_{rns} + 1, ly_{rns}, hy_{rns} + 1)$.

4.3 Node Restore

After MBR^n is inserted, any minimum bounding rectangles removed during node reduction are restored in the reverse order of their removal. The outcome is either a completely restored node or a set of nodes if a split is performed. This depends on the node validity following the replacement of each removed node region.

Each removed node region has a direction of north, south, east, or west, which is determined during node reduction. This direction is used to restore the node region with respect to $(lx_{rns}, hx_{rns} + 1, ly_{rns}, hy_{rns})$. Recall that after finding a location to insert MBR^n, one or both of the upper node region boundaries hx_{rns} and hy_{rns} are increased. The node region boundary hx_{rns} increases by one when either an east or east-swap insert occurs. The node region boundary hy_{rns} increases by one when either a north or north-swap occurs. Both hx_{rns} and hy_{rns} increase by one when either a northeast or northeast swap occurs. The lower node region boundaries lx_{rns} or ly_{rns} are not affected during node reduction because the node region never increases to the south or west.

When hx_{rns} and hy_{rns} increase, potential overlap problems arise between the node region $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns})$ and some node region in the reduction stack, namely the east and north node regions. The resulting cases for restore include north, north with overlap, east, east with overlap, south and west. These are depicted in Figure 7.

The east node region cases are presented below. The remaining cases are handled similarly.

1. Restoring an east node region (lx_e, hx_e, ly_e, hy_e) that does not overlap $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns})$. The east node region is replaced in its original location and the reduced node region is updated as follows:

   - the upper x boundary for the reduced node region is set to the upper x boundary for the east node region $(hy_{rns} = hy_e)$,
   - the upper y boundary for the reduced node region is set to the upper y boundary for the east node region if the latter is higher $(hy_{rns} = hy_e \ if \ hy_{rns} < hy_e)$, and
   - the lower y boundary for the reduced node region is set to the lower y boundary for the east node region if the latter is lower $(ly_{rns} = ly_e \ if \ ly_{rns} > ly_e)$.

2. Restoring an east node region (lx_e, hx_e, ly_e, hy_e) that overlaps $(lx_{rns}, hx_{rns}, ly_{rns}, hy_{rns})$. The east node region is shifted one column east from its original position when it is replaced and the reduced node region is updated as follows:

   - the upper x boundary for the reduced node region is set to the upper x boundary for the east node region plus one to account for the overlap column $(hx_{rns} = hx_e + 1)$,

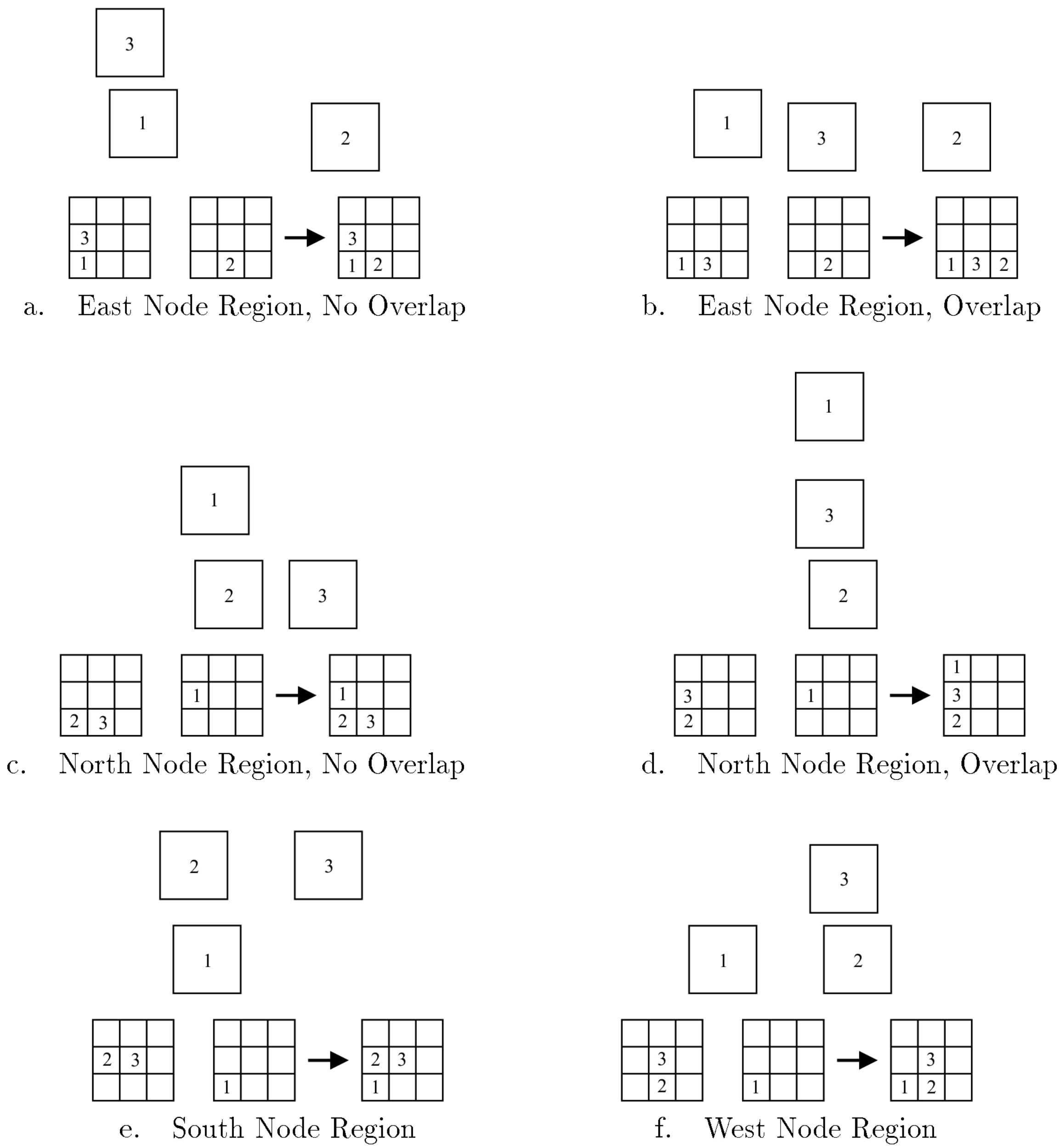


Figure 7: Node Restore Cases

- the upper y boundary for the reduced node region is set to the upper y boundary for the east node region if the latter is higher ($hy_{rns} = hy_e \; if \; hy_{rns} < hy_e$), and
- the lower y boundary for the reduced node region is set to the lower y boundary for the east node region if the latter is lower ($ly_{rns} = ly_e \; if \; ly_{rns} > ly_e$).

An overflow and validity test is performed after replacing a node region. If the node is valid and no overflow occurs, then node restoration continues with the next removed node region. If the node is invalid or overflow occurs, the node region is removed, and the partially restored node with the removed node regions undergo a reduction split (Section 4.6). For each node in the resulting set, a new minimum bounding rectangle that references it is inserted into the original parent node using node reduction. Each insertion is propagated to the root. Node restore continues until either all node regions are restored or a split occurs.

4.4 Node Adjustment

After removing minimum bounding rectangles from a node during the insertion path update or creating a new node during a split, it is necessary to relocate minimum bounding rectangles so that empty rows and columns are removed and empty locations within the object are minimized.

The relocation strategy performs a minor optimization by relocating minimum bounding rectangles located in the first row and column. For each minimum bounding rectangles in the first row, it is moved one column towards the origin (i.e. west) if the location directly west is empty and all remaining locations in the same column (i.e. north) are empty. For each minimum bounding rectangle in the first column, it is moved one row towards the origin (i.e. south) if the location directly south is empty, and all remaining locations in the same row (i.e. east) are empty.

4.5 Updating Insertion Path

After the node is restored or a split is performed, the minimum bounding rectangles along the insertion path are updated. In addition, the parent minimum bounding rectangles are inserted for new nodes formed from a split. At each level of the insertion path, a direct update of the minimum bounding rectangle is performed on the first visit to the level. This is followed by a validity test. If the node is invalid, the minimum bounding rectangle is removed and reinserted using node reduction and node restoration. If a second validity test fails, then a split at the non-leaf level is performed.

The simplest updates occur when the leaf node is fully restored and all nodes on the insertion are updated without causing splits. Beginning with the non-leaf node that references the leaf node, the minimum bounding rectangles along the insertion path are updated until the root is reached. If a split occurs at a non-leaf level, a new parent minimum bounding rectangle is inserted for each new node using the same strategy as that for leaf-level parent minimum bounding rectangles. Each split is propagated up the tree before any remaining minimum bounding rectangles are inserted on that level. If the root is split, a new root is created.

4.6 Splitting

The 2DR-tree uses a forced split, a reduction split, or a combination of the two. The forced split is used when overflow occurs but no node regions are removed during node reduction. This takes place when a base case involving multiple minimum bounding rectangles is reached during the first iteration of a node reduction.

A forced split is performed as follows. First, the node region that contains all objects including MBR^n is identified. Second, the midpoint location along the axis with the longest range in the node region is calculated. If the x-axis is chosen, the forced split is called an X-split. If the y-axis is chosen, the forced split is called a Y-split. Third, a new node is created and objects are moved to it. For an X-split, all objects from the split point and east are moved to the new node. For a Y-split, all objects from the split point and north are moved to the new node. In either case, the resulting nodes are valid so no validity test is necessary. Finally, if overflow still occurs in either new node, another forced split is applied to the node.

The reduction split is used when either overflow or node invalidity occurs and node regions are removed during node reduction. It is performed by assigning each node region in the reduction stack to its own node. No validity test is performed since each node region is valid.

A combination of a reduction split and a forced split is used when node regions exist in the reduction stack, but overflow exists in the resulting node region after its remaining objects are aligned at $(0, 0)$. A forced split is performed to split the partially restored node and each remaining node region is assigned to its own node.

5 Deletion

This section presents the 2DR-tree deletion strategy, which exploits both the existing spatial relationships to locate an object and the insertion techniques of node reduction, node restoration, and splitting to update minimum bounding rectangles along the search path. The object to be deleted is located using a binary search strategy. After the object is deleted, the search path is updated by removing unneeded nodes, replacing minimum bounding rectangles, and collapsing the root as necessary. The binary search, search path update, and root collapse strategies are presented below. The node reduction, node restoration, and splitting strategies are discussed in the previous section on insertion.

5.1 Binary Search

Beginning at the root, a binary search is performed in each node encountered on the search path. First, the node region $(lx_{rt}, hx_{rt}, ly_{rt}, hy_{rt})$, which contains all locations in the root is identified. Second, the partitioning dimension is identified. The dimension with the longest range of index values in the node region is chosen. Third, the node region is partitioned through the midpoint of the chosen dimension. A partition through the x-axis at midpoint cx_{rt} produces $(lx_{rt}, cx_{rt} - 1, ly_{rt}, hy_{rt})$ and $(cx_{rt}, hx_{rt}, ly_{rt}, hy_{rt})$. A partition through the y-axis through the midpoint cy_{rt} produces $(lx_{rt}, hx_{rt}, ly_{rt}, cy_{rt}-1)$ and $(lx_{rt}, hx_{rt}, cy_{rt}, hy_{rt})$. For each new node region, a minimum bounding rectangle is then created which includes all objects in it.

Each minimum bounding rectangle is tested to see if it encloses the search object. If enclosure occurs, the node region undergoes further binary partitioning until either no more enclosure occurs or the node region contains one minimum bounding rectangle. If the minimum bounding rectangle encloses the search region, the search continues in the corresponding subtree.

If a leaf node is reached, the same binary search is performed. If a node region containing one object is reached, the object is tested for equality with the search region and is returned as part of the result if they are equal.

5.2 Updating Search Path

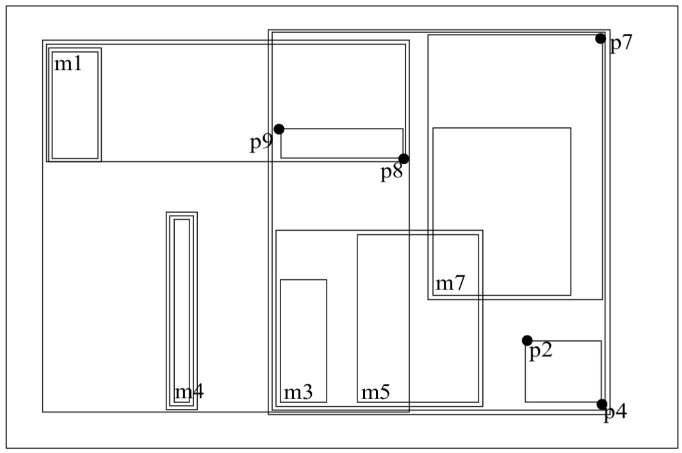
After deleting an object, several updates occur. First, any nodes no longer needed are removed, starting at the leaf node that contained the object. If the leaf node is empty because its only object was deleted, then the node and its parent minimum bounding rectangle are deleted. This continues along the search path until a node is reached that still contains objects. Second, the minimum bounding rectangles along the remaining search path are updated. The old minimum bounding rectangles are moved one at a time and an updated minimum bounding rectangle is inserted using node reduction and restoration. If a split occurs, it is handled in the same manner as for insertion. Finally, the root node is collapsed if the root contains only one minimum bounding rectangle, or if all of its immediate descendents contain only one minimum bounding rectangle. In the former case, the immediate descendent of the root node becomes the root. In the latter case, the immediate descendents are deleted, and their minimum bounding rectangles are assigned to the same location in the root that contained their parent minimum bounding rectangle.

6 Example

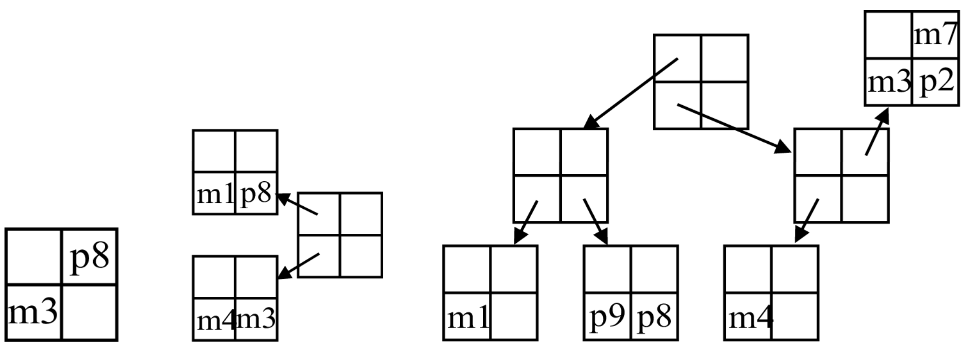
This section demonstrates 2DR-tree insertion and deletion. An order 2*2 2DR-tree is created using the example point and object data set (Figure 8.a), which is a subset of the running example from Günther and Gaede [GG98]. The object insertion order is p8, m3, m1, m4, p9, p2, m7, m5, p4, and p7. Using this 2DR-tree, the objects p7, m7 and m4 are then deleted from the tree. For both examples, subtrees are displayed using the objects it contains, and one or more levels of parentheses to indicate its height. For examples, (p4,p7) represents a leaf node (i.e.: height 0), while ((p4,p7),(m5)) represents a subtree of height 1, containing two minimum bounding rectangles that reference (p4,p7) and (m5), respectively.

6.1 Insertion

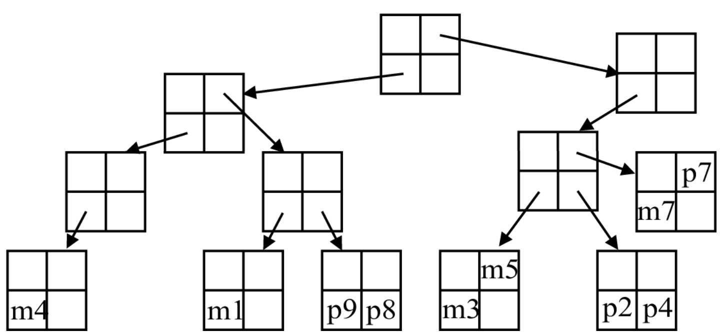
Point p8 is inserted into the root first at location $(0,0)$. Object m3 is then inserted. The root contains one object only - p8 - so the spatial relationship between m3 and p8 is compared. Since p8 is located northeast of m3, a northeast swap insert is performed. Object m3 replaces p8 at location $(0,0)$



a. Data Set



b. p8,m3 c. m1,m4 d. p9,p2,m7



a. m5,p4,p7

Figure 8: 2DR-tree Insertion

and p8 is inserted at location $(1, 1)$. Figure 8.b shows the 2DR-tree after inserting p8 and m3.

Object m1 is inserted next. Multiple objects exist in the root, so the spatial relationship between m1 and the node space of the root is compared to determine if it satisfies a multiple-object base case. Object m1 is located northwest of the node space and the slope between the centroids of m1 and the node space is greater than -1, so none of the multiple-object cases are satisfied. Therefore, the node space is partitioned through its y-axis. Since m1 is located north of the partition, the object m3 is removed and placed on the reduction stack. The node space now contains p8. Object m1 is located northwest of m8 and the slope between the objects is greater than -1, so an east swap insert is attempted. However, an overflow of the node boundary occurs, so a reduction split is performed to produce new nodes (m1,p8) and (m3). A new root node is created and minimum bounding rectangles that will reference the new nodes in their parent (i.e. root) are inserted. The minimum bounding rectangle for (m1,p8) is inserted at $(0, 0)$. The minimum bounding rectangle for (m3) is northwest of the minimum bounding rectangle for (m1,p8) with a slope less than -1 so it is inserted at $(0, 1)$.

Object m4 is inserted next. The tree now has two levels so a search for the appropriate leaf node is performed. A greedy search of the root node compares the area increase required at $(0, 0)$ with that required at $(0, 1)$ only because $(1, 1)$ and $(1, 0)$ contain no minimum bounding rectangles. Location $(0, 0)$ has the minimum area increase so the search continues in the corresponding subtree, which is a leaf node (m3). Object m4 is located northwest of m3 and the slope between them is greater than -1. Therefore, an east swap is performed. Object m3 is swapped with m4 and m3 is inserted at location $(1, 0)$. The existing minimum bounding rectangle that referenced (m3) in the parent node is then deleted, and an updated minimum bounding rectangle is inserted. After comparing the minimum bounding rectangle for (m4,m3) with that for (m1,p8), the minimum bounding rectangle for (m4,m3) is inserted at $(0, 0)$ using a north swap insert. Figure 8.c shows the 2DR-tree after inserting m1 and m4.

When point p9 is inserted next into (m1,p8), overflow of the node boundary occurs. A reduction split produces (p9,p8) and (m1). A new minimum bounding rectangle for (p9,p8) is inserted at $(1, 1)$ in the root node. When a new minimum bounding rectangle for (m1) is inserted, overflow of the root boundary occurs so a reduction split produces ((m1),(p9,p8)) and ((m4,m3)). After a new root is created, a new minimum bounding rectangle for ((m1),(p9,p8)) is inserted at $(0,0)$. The minimum bounding rectangle for ((m4,m3)) is swapped with the minimum bounding rectangle for ((m1),(p9,p8)) and the latter is inserted at $(0,1)$.

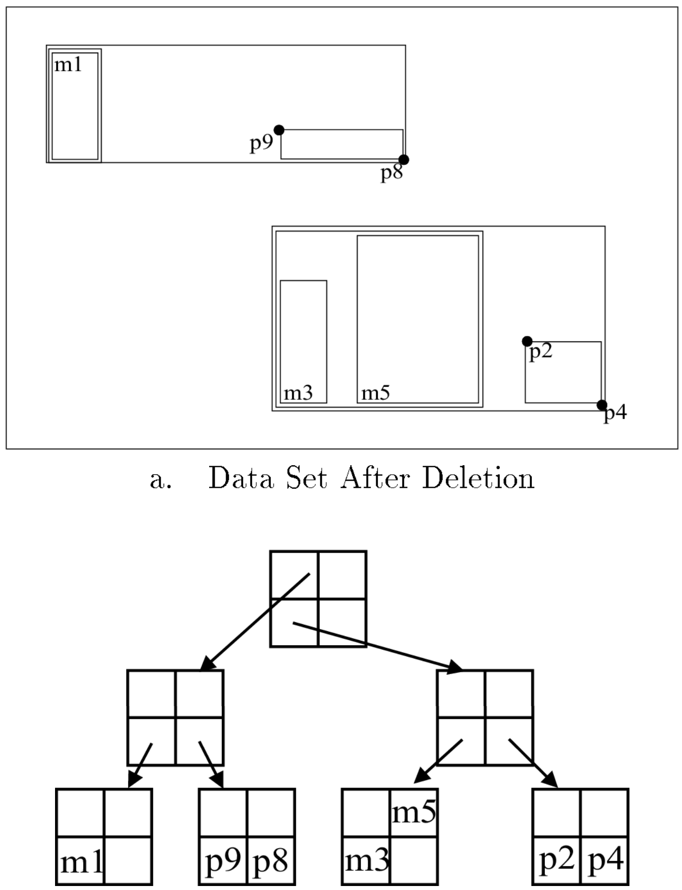
The objects p2 and m7 are then inserted. Point p2 is inserted into (m4,m3), which causes a split to produce (m4) and (m3,p2). Minimum bounding rectangles are successfully inserted at all non-leaf levels. When object m7 is inserted, node region $(0,0)$ is generated and m7 is inserted north of p2 at location $(1,1)$. When node region $(0,0)$ is put back, the validity test passes, and the node is restored. All remaining updates along the insertion path are successful. Figure 8.d shows the 2DR-tree after inserting p9, p2, and m7. The objects m5, p4, and p7 are then inserted. Figure 8.e shows the final 2DR-tree.[1]

6.2 Deletion

The point p7 is deleted first. The binary search begins with the node region for the root. Since the x-range and y-range are equal, the node region is partitioned through the x-axis to produce $((0,0),(0,1))$ and $((1,0),(1,1))$. A minimum bounding rectangle is found for the node region $((0,0),(0,1))$ but it does not enclose p7. A minimum bounding rectangle is found for $((1,0),(1,1))$. It encloses p7 so the node region is partitioned through the y-axis to produce $((1,0))$ and $((1,1))$. The node region $((1,0))$ contains no minimum bounding rectangles. The node region $((1,1))$ contains one minimum bounding rectangle that encloses p7, leading to a binary search at level 1.

Once again the x-range and y-range are equal so the node region is partitioned through the x-axis to produce $((0,0),(0,1))$ and $((1,0),(1,1))$. A minimum bounding rectangle is found for the node region $((0,0),(0,1))$. It encloses p7 so the node region is partitioned through the y-axis to produce $((0,0))$ and $((0,1))$. The node region $((0,0))$ contains a minimum bounding rectangle that encloses p7, leading to a binary search at level 2. The binary search will determine that the minimum bounding rectangle in $((1,1))$ encloses p7, which takes the search to a leaf node. Using a binary search, p7 is found in $((1,1))$ and is deleted.

[1] Care must be taken to follow the pointers correctly as their origin respects the objects spatial locality but the index represented in Figure 8.d and 8.e are somewhat rotated to ensure they fit on the diagram.



a. Data Set After Deletion

b. After Deleting p7,m7,m4

Figure 9: 2DR-tree Deletion

The search path is now updated. Since the leaf node contains objects, it cannot be deleted, so the minimum bounding rectangles along the search path are updated. The existing minimum bounding rectangle that references the leaf in its parent node is deleted and a new parent is inserted into location $(1,1)$. Propagating upwards, new minimum bounding rectangles are inserted into locations $(0,0)$ in level 1 and $(1,0)$ in the root. No tree collapsing is performed since both the root and one of its immediate descendents has multiple entries.

Object m7 is deleted next. The same search path that was found to locate p7 is found when locating m7. However, after m7 is deleted, the leaf is empty so it is deleted along with the corresponding minimum bounding rectangle in its parent node in level 3. The remaining minimum bounding rectangles are inserted using node reduction into location $(0,0)$ in level 1 and $(1,0)$ in the root. No tree collapsing is performed since both the root and one of its immediate descendents have multiple entries.

Object m4 is now deleted. The search path to locate m4 is $(0,0)$ in all levels. After m4 is deleted, the nodes on the search path at levels 3 and 2 are deleted along with the corresponding minimum bounding rectangle in level 1. Using node reduction, the remaining minimum bounding rectangle on the search path is inserted into $(0,1)$. Since all descendents of the root have one entry each, the tree is collapsed. The single entry in each descendent is moved to its proper location in the root and the extra nodes are deleted. Figure 9.b shows the 2DR-tree after deleting p7, m7, and m4 in addition to collapsing the tree.

7 Evaluation

This section presents the methodology and results of a performance evaluation on the 2DR-tree. The evaluation methodology is presented first followed by a discussion of the experimental results.

7.1 Methodology

The rationale for the insertion and deletion performance evaluation is to observe the behaviour of the 2DR-tree for different node sizes, numbers of objects, and data distributions. The 2DR-tree is implemented using Java on Dell Precision WorkStation 530 MT running Windows 2000. The object sets are stored in databases managed by IBM DB2.

The performance evaluation uses several object sets. UNIF100, UNIF500, UNIF1000, UNIF2000, UNIF4000, UNIF6000, UNIF8000 and UNIF10000 contain 100, 500, 1000, 2000, 4000, 6000, 8000 and 10000 squares, respectively. Each square covers approximately 1% of the "space". Each set of objects is uniformly distributed. Not considering overlap, each set covers approximately 90-100% of the space. With between 51-53% overlap, each set covers approximately 67-75% of the space. All uniformly distributed object sets were created by copying objects from a subspace of UNIF10000, instead of either randomly generating objects in a smaller space or randomly copying objects from the entire space for UNIF10000. The reason for this is to be able to accurately track increases in coverage, overcoverage, and overlap as the size of the object set size increases. Randomly generating objects in a smaller space will result in different objects being covered altogether, while randomly selecting objects for the smaller sets will cause more overcoverage than is necessary for few objects. SKEW500 contains 500 squares with an exponential (i.e.: skewed) distribution. This set was generated separately, but the squares are the same dimension, and achieve the same non-overlap coverage as UNIF500, although more overlap exists due to the distribution.

Each test run will construct 1000 trees using a random sort of the object set. For each test run, the tree height, number of nodes, average space utilization, coverage, overcoverage, overlap, average number of disk accesses per insert/delete, and the average number of splits per insert/delete are recorded. Coverage is calculated as the total area of all minimum bounding rectangles. Overlap is calculated as the sum of overlap on each level, with overlap on each level being the sum of overlap between all pairs of objects. Overcoverage is calculated as the sum of overcoverage on each level. Overcoverage for one level of the tree is calculated as coverage of the minimum bounding rectangles minus the coverage (with overlap being taken into consideration) of the minimum bounding rectangles in its immediate descendent nodes. Overcoverage is a calculation of "wasted space" in the minimum bounding rectangles. Each disk access retrieves one node. It is assumed that each page from secondary storage stores the information for one node, independent of node size. It is also assumed, pessimistically, that with the exception of the root node and new nodes produced from a split, each node is retrieved every time it is read. In the latter case, the minimum bounding rectangles required for updating are generated immediately after creating the node so retrieving new nodes resulting from a split is not required.

The insertion performance is evaluated using three sets of test runs. The first set of test runs perform a full range of test to determine the effect of the number of objects inserted on all performance parameters. Each set of test runs will insert 100, 500, 1000, 2000, 4000, 6000, 8000 and 10000 objects, respectively. A 5x5 node size and uniform distribution of data are used. The second set of test runs focus on the effect of node size on tree height, average number of disk accesses per insert, and space utilization. Each set of test runs uses 3x3, 5x5, 7x7, and 10x10 nodes, respectively. A uniformly distributed set of 500 objects is used for all second-set test runs. The third set of test runs focus on the effect of data distribution on coverage, overcoverage, and overlap. Each test run uses 500 uniformly and exponentially distributed objects respectively. A 5x5 node size is used.

The deletion performance is also evaluated using three sets of test runs. Each test run will take each of the 1000 trees created during the insertion test and delete every tenth item, with the exception of the first set of test runs. The first set of test runs uses the trees for the 100, 500, 1000 and 10000 object sets and deletes 10, 50, 100, and 1000 objects, respectively. The second and third set of test runs will delete 50 objects from each tree. For all test runs, the same performance parameters as for insertion are recorded. The goal for all test runs is to identify any significant difference in a performance parameter after deleting a selection of objects.

7.2 Results and Discussion

7.2.1 Insertion and Object Set Size

Table 1 shows the averages for each run when varying the number of objects inserted into the 2DR-tree. The average number of seeks per insert is less than three times the average tree height in all cases. This includes the number required to find the appropriate leaf node so updating takes the remaining seeks. The average number of seeks for updating is more than one times the average tree height due to the average of approximately one split per insertion occurring. This is significant since splits can be triggered by many situations in the 2DR-tree, including full nodes, objects falling outside of node bounds, and invalidity during a node restore. The results show that splits are not a huge factor for 2DR-tree insertion.

Figure 10 shows the effect of the number of objects inserted on both the tree height and the average number of disk accesses required for inserting

#Objects	#Nodes	Height	Sp.Util.	Coverage
100	127.97	8.10	1.78	128,906.74
500	654.06	12.31	1.76	1,233,595.22
1,000	1,312.53	14.19	1.76	3,232,169.34
2,000	2,619,52	16.23	1.76	8,784,155.25
4,000	5,244.69	18.26	1.76	23,448,427.79
6,000	7,867.63	19.47	1.76	41,887,513.70
8,000	10,470.28	20.44	1.76	63,752,968.22
10,000	13.077.67	21.00	1.76	88,158,132.24

#Objects	Overcoverage	Overlap	#Seeks/Ins	#Splits/Ins
100	15,801.81	13.070.12	17.82	1.04
500	138,768.90	129,630.80	30.74	1.12
1,000	355,257.04	340,590.69	36.74	1.13
2,000	959,547.75	935,355.26	43.11	1.13
4,000	2,535,300.42	2,494,116.09	49.59	1.13
6,000	4,525,004.07	4,465,319.16	53.43	1.13
8,000	6,862,775.80	6,792,707.51	56.28	1.13
10,000	9,517,985.63	9,432,726.01	58.20	1.13

Table 1: Averages From Insert Set 1

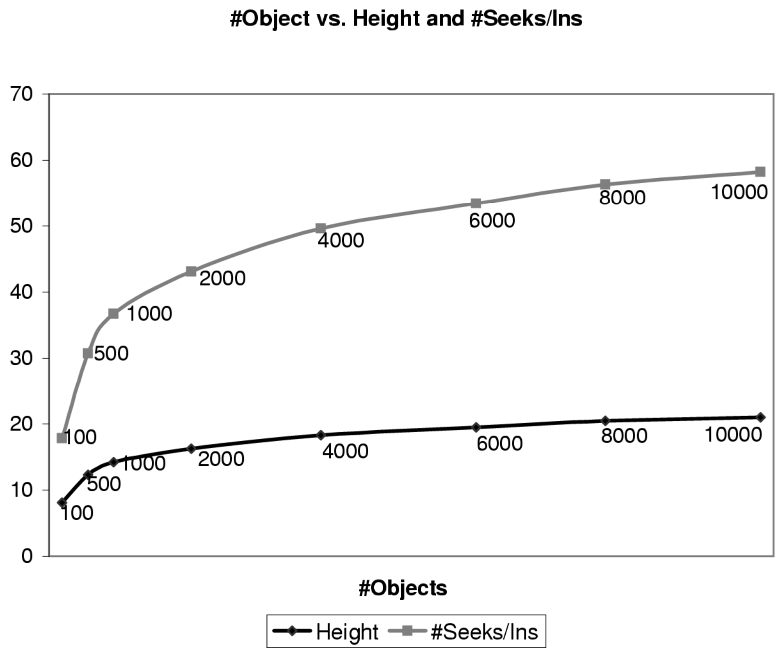


Figure 10: Object Set Size vs. Height and #Seeks/Insert

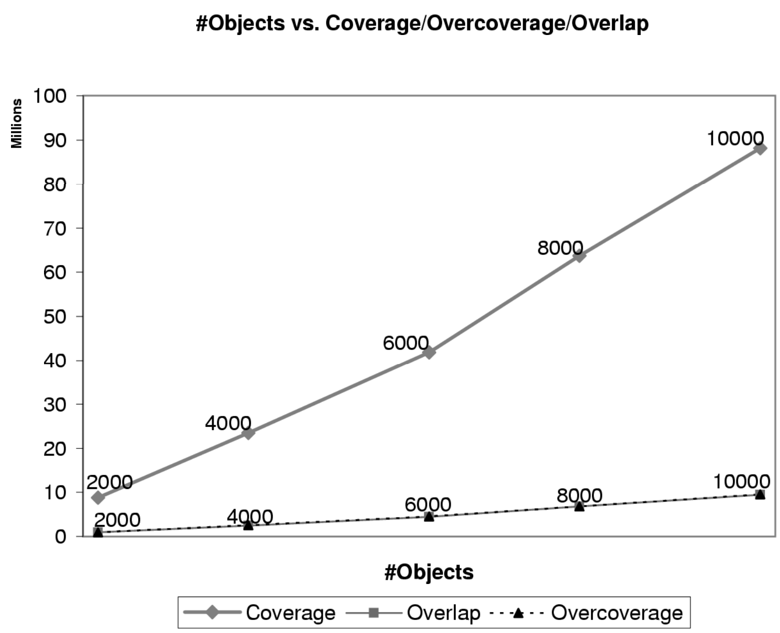


Figure 11: Object Set Size vs. Coverage/Overlap/Overcoverage

Node Size	#Nodes	Height	Sp.Util.	Coverage
3x3	700.01	12.53	1.71	1,224,639.62
5x5	654.06	12.31	1.76	1,233,595.22
7x7	653.10	12.38	1.77	1,244,505.89
10x10	652.99	12.39	1.77	1,254,126.55

Node Size	Overcoverage	Overlap	#Seeks/Ins	#Splits/Ins
3x3	133,937.22	125,119.97	32.12	1.22
5x5	138,768.90	129,630.80	30.74	1.12
7x7	139,285.40	130,155.75	30.87	1.11
10x10	140,921.96	131,741.72	30.87	1.11

Table 2: Averages From Insert Set 2

a new object. Initially, the tree grows in height quickly but growth slows significantly as more objects are inserted. The same occurs for the number of disk accesses.

Figure 11 shows the effect of the number of objects inserted on the coverage, overlap and overcoverage. Results show that the coverage increases linearly as the number of objects increase. In addition, the rate of increase in overlap and overcoverage is significantly lower as the number of objects increase. Coverage includes the object coverage, while overlap and overcoverage are only calculated for non-leaf nodes. One reason for the lower growth in overlap and overcoverage is the ability of the 2DR-tree to "cluster" objects located close together as the number of objects increase, which reduce both the overlap and the wasted space in non-leaf minimum bounding rectangles.

7.2.2 Insertion and Node Size

Table 2 shows the averages for each run when varying the node size. Results show that the tree height, number of nodes, and the average space utilization do not significantly change as the node sizes increase. This contradicts the expectation that the tree height and number of nodes would significantly decrease, while the space utilization would significantly increase. However, one option to explore is the following. Currently, the 2DR-tree requires a disk access for each node, whether it is a 2x2 or a 10x10. If allocating larger nodes does not decrease height and increase space utilization, then a page assignment strategy that retrieves several smaller nodes in one disk access

Distribution	#Nodes	Height	Sp.Util.	Coverage
Uniform	654.06	12.31	1.76	1,233,595.22
Exponential	652.63	12.31	1.77	638,430.98

Distribution	Overcoverage	Overlap	#Seeks/Ins	#Splits/Ins
Uniform	138,768.90	129,630.80	30.74	1.12
Exponential	64,157.58	67,130.08	30.71	1.10

Table 3: Averages From Insert Set 3

must be considered. Allocating larger nodes will require more disk accesses, and given that the number used in the 10x10 tree is not significantly lower than the number used in the other trees, it may be an option not to use large nodes at all.

7.2.3 Insertion and Data Distribution

Table 3 shows the averages for each run when varying the distribution of the data set. The results show a significant difference in coverage, overcoverage, and overlap. The surprising result is that when indexing exponentially distributed data, the 2DR-tree achieves significantly, almost 50%, lower coverage and overlap, and 54% lower overcoverage. The height, number of nodes, and space utilization are not a factor in this because they are not significantly different between the data distributions. One reason for this is the following. After many insertions, "chains" that consist of many non-leaf nodes that lead to one node with few objects - possibly one - start to appear. An advantage to chains is that outliers are separated from a cluster of objects, which reduces the coverage, overcoverage, and overlap of minimum bounding rectangles.

7.2.4 Deletion

Tables 4, 5, and 6 show the results after deleting every 10th item out of each tree creating during insertion. The main goals of the deletion experiments are the following. The first is to see how significant splitting and seeking is to the deletion process. The second is to observe any significant changes in height, space utilization, coverage, overcoverage, and overlap.

#Objects	#Nodes	Height	Sp.Util.	Coverage
90	128.26	8.23	1.70	125,677.77
450	656.46	12.49	1.69	1,209,182.42
900	1,317.31	14.38	1.68	3,171,132.08
9000	13.132.17	21.26	1.69	86,788,433.81

#Objects	Overcoverage	Overlap	#Seeks/Del	#Splits/Del
90	15,674.41	12,237.95	20.50	0.03
450	136,435.15	123,735.55	34.88	0.03
900	349,176.35	327,206.99	43.26	0.03
9000	9,332,449.38	9,154,086.98	91.10	0.03

Table 4: Averages From Delete Set 1

Node Size	#Nodes	Height	Sp.Util.	Coverage
3x3	700.34	12.71	1.64	1,200,686.35
5x5	656.46	12.49	1.69	1,209,182.42
7x7	655.30	12.53	1.69	1,217,572.39
10x10	655.57	12.55	1.69	1,228,114.60

Node Size	Overcoverage	Overlap	#Seeks/Del	#Splits/Del
3x3	132,229.76	119,776.29	35.09	0.03
5x5	136,435.15	123,753.55	34.88	0.03
7x7	136.982.40	124.269.60	35.09	0.03
10x10	138,881.18	126.139.44	35.31	0.03

Table 5: Averages From Delete Set 2

Distribution	#Nodes	Height	Sp.Util.	Coverage
Uniform	656.46	12.49	1.69	1,209,182.42
Exponential	654.95	12.55	1.69	621,735.51

Distribution	Overcoverage	Overlap	#Seeks/Del	#Splits/Del
Uniform	136,435.15	123,753.55	34.88	0.03
Exponential	61,437.16	62,858.33	37.45	0.03

Table 6: Averages From Delete Set 3

In all cases the average number of splits per deletion is approximately 0.3, which indicates that splitting is not a significant factor. However, in all cases the average number of seeks required to delete an object and perform updates is about three times the height. Since few splits per delete occur on average, about two times the height occur from searching for this object. This means that, although the search used involves enclosure, enough overlap exists that multiple paths must be searched to find the object.

For all cases, there are no significant changes as a result of deleting 10% of the objects from the tree. For both the first and second set of test runs, the coverage, overcoverage, and overlap decrease slightly in all test runs. For the third set of test runs, the coverage, overcoverage, and overlap decrease slightly for the uniform object set, but increase slightly for the exponential object set. Therefore, object deletion does not adversely affect the 2DR-tree.

8 Conclusion

This paper presents the 2DR-tree, a spatial access method that preserves spatial relationships between all objects. This is achieved by using nodes that are the same dimension as the object set. This structure supports both binary and greedy searching in multiple dimensions. Insertion uses a node reduction technique to insert a new object, and uses the node regions already formed during node reduction in a low-overhead reduction splitting strategy.

Experimental results show that the 2DR-tree is ideal for larger objects sets with respect to tree height. The average number of disk accesses and split per insert are reasonable. In addition, it is ideal for a dynamic skewed data set, which achieve lower coverage, overcoverage, and overlap than a dynamic, uniformly distributed data set.

Future research directions include the following. The first is to improve both the average space utilization and the amount of searching for objects during deletion. Several optimization techniques will be explored. The second is to develop an algorithm for bottom-up tree construction that can be applied to static data sets. The third is to develop algorithms for other types of spatial searches, such as adjacency, nearest neighbour, spatial join, and direction queries. The fourth is to perform an empirical evaluation of the 2DR-tree versus other proposed SAMs. Finally, the 2DR-tree will be extended to three dimensions.

References

[AEUB04] Alberta Energy and Utilities Board. Alberta Geographical Survey GIS, http://www.ags.gov.ab.ca/gis/gis_and_mapping.shtml. (visited May 2004), 2004.

[AWYE97] 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 on Spatial Databases SSD '97*, number 1262 in Lecture Notes in Computer Science, pages 350–73. Springer-Verlag, 1997.

[Bay97] R. Bayer. The universal B-tree for multidimensional indexing: general concepts. In *Worldwide Computing and Its Applications, WWCA '97*, number 1274 in Lecture Notes in Computer Science, pages 198–209. Springer-Verlag, 1997.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[Ben79] J.L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transacations on Software Engineering*, 5:333–340, 1979.

[Bio04] Research Collaboratory For Structural Bioinformatics. Protein data bank, http://www.rcsb.org/pdb. (visited May 2004), 2004.

[BKK96] S. Berchtold, D. 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*, pages 28–39, 1996.

[BKSS90] 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 International Conference on Management of Data*, pages 322–31, 1990.

[BM74] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–188, 1974.

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 426–435, 1997.

[FB74] R.A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[FBF77] J.H. Friedman, J.L Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3:209–26, 1977.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, 1998.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[Hin85] K. Hinrichs. Implementation of the grid file: design concepts and experience. *BIT*, 25:569–92, 1985.

[HSW89] A. Henrich, H.-W. Six, and P. Widmayer. The LSD-tree: spatial access to multidimensional point and non-point objects. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 45–53, 1989.

[KF94] I. Kamel and C. Faloutsos. Hilbert R-tree: an improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–9, 1994.

[Kou00] N. Koudas. Indexing support for spatial joins. *Data and Knowledge Engineering*, 34:99–124, 2000.

[NHS81] J. Nievergelt, H. Hintenberger, and K. Sevcik. The grid file: an adaptable symmetric multikey file structure. In *Proceedings of the Third ECI Conference*, number 123 in Lecture Notes in Computer Science, pages 236–51. Springer-Verlag, 1981.

[NHS84] J. Nievergelt, H. Hintenberger, and K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9:38–71, 1984.

[OM84] J Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–90, 1984.

[Ore89] J. Orenstein. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 294–305, 1989.

[RL85] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 17–31, 1985.

[Rob81] J.T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[RSV01] P. Rigaux, M. Scholl, and A. Voisard. *Spatial databases: with application to GIS*. Morgan-Kauffman, 2001.

[Sam90] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley, 1990.

[SK96] K.C. Sevcik and N. Koudas. Filter trees for managing spatial data over a range of size granularities. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 16–27, 1996.

[SRF87] 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 Data Bases*, 1987.