# A Survey on Multidimensional Access Methods

*Hee-Kap Ahn*
*Nikos Mamoulis*
*Ho Min Wong*

UU-CS-2001-14
May, 2001

# A Survey on Multidimensional Access Methods*

Hee Kap Ahn†, Nikos Mamoulis§ and Ho Min Wong§
†Institute of Infomation and Computing Sciences, Utrecht University, The Netherlands,
heekap@cs.uu.nl
§Dept. Computer Science, Hong Kong University of Science and Technology, Hong Kong
{mamoulis, mikew}@cs.ust.hk

**Abstract**

The extraordinary format of spatial data and the fact that there is no straightforward mapping of spatial objects from the multidimensional space to the 1-dimensional space, stimulated various researchers during the past two decades to develop *multidimensional access methods* that facilitate efficient indexing of spatial objects in large databases. This survey paper tries a classification of existing multidimensional access methods, according to the types of data they are most suitable for (points or objects with spatial extent), their structure (hierarchical or flat), and their performance over spatial queries. Most of this work is based on an excellent survey paper[Gaed97]

# 1. Introduction

To deal with multidimensional data, it is desirable to design spatial data management systems in which spatial operations can be performed fast. Towards this task, the database community has devoted considerable attention to spatial data management. The main motivation originated from an increasing number of computer applications such as VLSI CAD and cartography.

Multidimensional data include points, line segments, rectangles, polygons, regions, volumes, and polyhedra in 2D, 3D or Higher. Spatial databases contain multidimensional data with explicit knowledge about objects, their extent, and their position in space. The objects are represented in some vector-based format, and their relative position may be explicit or implicit.

Several Multidimensional Access Methods, some general purpose and some application specific, that support search operations in spatial databases have been proposed and evolved for about 30 years. In this paper, we discuss the most prominent data structures and present possible taxonomies of multidimensional access methods.

The remainder of this paper is organized as follows. Section 2 discusses the basic properties of spatial data and clarifies the requirements a multidimensional data access method should meet. We also discuss queries on spatial data, and classify access methods into Point Access Methods and Spatial Access Methods. Section 3 gives an overview of some traditional, but most prominent spatial data structures; the grid file and its variants, the quadtree and its variants, the k-d tree and its variants, and the R-trees. We discuss comparative studies and performance analyses of spatial data structures in Section 4. Finally, section 5 concludes the paper.

# 2. Spatial Data

## 2.1 Characteristics of spatial data

Spatial data are considered as special kind of data, as they have several characteristics that call for non-standard database management methods for their handling. These characteristics can be summarized as follows:

a) Spatial objects have *complex structure*. A single point, or a set of several arbitrarily distributed polygons can characterize a spatial data object, as well. Relational database tuples with fixed size are not suitable to store such variety of data formats. As a result, spatial operations (e.g. intersection, union) are computationally *more expensive* than standard RDBMS operations.

* This work was conducted for the purposes of COMP630c, "Spatial, Image and Multimedia Databases," Oct. 1997

b) Spatial data is often *dynamic*. This characteristic of spatial data requires data structures robust to frequent insertions, deletions and updates of objects.
c) Spatial databases tend to be *large*. The number of objects in a geographic map, or a VLSI circuit often demands several gigabytes of storage. The integration of secondary memory in spatial data structures is therefore a must.
d) There is *no standard spatial algebra*. No set of standard spatial operators have been defined, as they usually depend on the application domain of the specific spatial database.
e) Spatial operators are *not closed*. The intersection of two spatial objects, for instance, may return a set of points, lines or regions.

Another important characteristic concerning spatial data, is that as they are multidimensional, there exists no total ordering among spatial objects, that preserves spatial proximity[Gaed97]. That is, there is no way we can order spatial objects in a linear fashion, so that objects that are spatially close to each other in the two- or higher-dimensional space, are close to each other in the linear order. This characteristic makes it very difficult to apply traditional database indexing methods, such as B-trees or linear hashing to index spatial databases.

## 2.2 Requirements of spatial data access methods

The spatial data properties mentioned above make the design of efficient spatial data access methods a laborious and challenging task, that has to meet a variety of requirements, including:

a) *Dynamics.* As data objects can be inserted and deleted form a spatial database in any given order, data access methods should continuously keep track on these changes.
b) *Secondary/tertiary storage management.* Spatial access methods need to integrate efficiently secondary and tertiary storage.
c) *Support of several operations.* A broad range of operations on spatial data should be supported. Spatial data access methods should not focus on the performance over one operation (e.g. searching), with the cost of slow and ineffective other operations (e.g. deletion).
d) *Independence of the input data and insertion sequence.* The performance of an access method should not depend either on the kind of input data, or on the order in which they are inserted.
e) *Scalability.* Access methods should adapt well to database growth.
f) *Time and space efficiency.* Spatial access methods should operate fast, with a logarithmic worst-time performance, given any set of input data. The space indices used should be small and should guarantee a satisfactory space utilization.

## 2.3 Queries on spatial data

### *2.3.1 Spatial selection queries*

As noted above, there is no standard spatial algebra, or standard spatial query language. Query languages for spatial databases heavily depend on the application domain, as many standardization attempts have failed to cover all potential spatial query requirements. The result a spatial database query is usually a set of spatial objects that satisfy the properties of the query. Several kinds of queries can be applied on spatial databases including:

a) *Exact Match Query:* Find all database objects that have *exactly the same spatial extent* (i.e. spatial attributes) as the spatial query object o.
b) *Point Query:* Find all database objects that *overlap* the query point p.
c) *Window Query or Range Query:* Find all database objects that have *at least one common point* with a d-dimensional query window w.
d) *Intersection Query or Region Query or Overlap Query:* Find all database objects that have *at least one common point* with a query object o.
e) *Enclosure Query:* Find all database objects that *enclose* a query object o. An object a is said to enclose object b iff any point of a is part of object b.
f) *Containment Query:* Find all database objects that *are enclosed* by a query object o.

g) *Adjacency Query:* Find all database objects that *are adjacent* to a query object o. Two objects are said to be adjacent if they have common boundaries, but the one does enclose the other.
h) *Nearest Neighbor Query:* Find all database objects that *have a minimum distance* from a query object o. Distance between spatial objects is usually defined as the distance between their closest points.

### *2.3.2 Spatial Joins*

Besides spatial selection queries, where a set of objects that satisfy the query criteria are selected from a database relation, the spatial join is a common and very important operation on a spatial database. A relational θ-join of two relations $R_1$, $R_2$ on columns $i \in R_1$, $j \in R_2$, is called *spatial join*, if the i-th column of $R_1$ and the j-th column of $R_2$ are spatial attributes and θ is a spatial predicate[Günt93].

The most common spatial join operation is the *intersection join*, where θ is the intersection operator. Brinkhoff et al. in [Brin93] define the MBR-spatial-join as an intermediate-filter step to compute the intersection join between two relations. The MBR-spatial-join is the intersection join of the minimum d-dimensional rectangles (Minimum Boundary Rectangles) that contain the joined objects. The main idea on which they base their work is that if the MBRs of two objects do not intersect, they exact objects will not intersect either. This property can be used to filter out of a spatial join object pairs, by simply testing their MBRs intersection. Assuming that spatial objects in both joined relation columns are organized in $R^*$-trees[Beck90], they suggest several heuristic techniques which efficiently compute the MRB-spatial join, as far as CPU and I/O time costs are concerned.

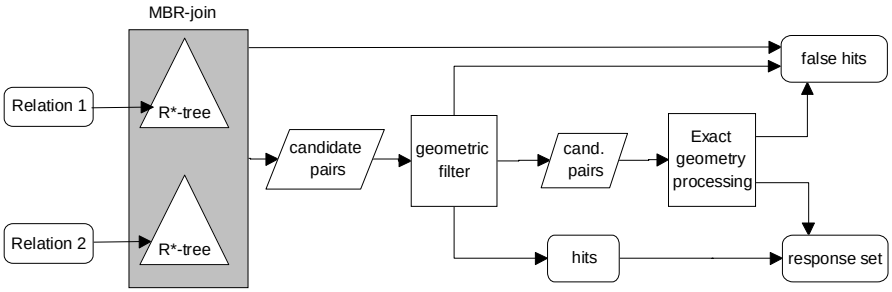


Figure 1: Multi-step processing of spatial joins [Brin94]

In [Brin94] the same research group propose an integrated solution that calculates the intersection join of two relations in three steps (see figure 1). At first, they apply algorithms from [Brin93] to calculate the MBR-join of the relations. The next step of the join process is to apply again cheap geometric filters that identify object pairs that do not definitely intersect (false hits) and object pairs that intersect for sure (hits), decreasing the number of candidate pairs that need further processing. To identify the false hits, a more precise convex approximation (e.g. convex hull) is calculated for each object in a candidate set, and a new intersection test is applied. Hits are filtered by doing the intersection test over the Maximum Enclosed Rectangles (MER) of objects in each candidate pair. The MER of an object is defined to be the maximum d-dimensional rectangle that can be enclosed into the object's spatial extent. If the MERs of two objects intersect then the objects intersect for sure. After the second step of the process, the remaining candidate pairs have to processed using expensive geometric algorithms (e.g. plane sweep) that apply on the exact spatial extent of the objects.

## 2.4 Classification of access methods

Gaede et al. [Gaed97] classify multidimensional data access methods into Point Access Methods (PAM) and Spatial Access Methods (SAM). PAM were primarily designed to perform spatial searches on point databases; databases that store only multidimensional points that do not have spatial extension. On the other hand SAM manage objects that, apart from their position in space, have spatial characteristics (shape). Such objects are lines, polygons, or higher-dimensional polyhedra.

### *2.4.1 Point Access Methods (PAM)*

Point access methods generally organize the point data in buckets, each corresponding to a disk page and to some sub-space of the universe. The buckets, usually rectilinear, are indexed by either flat or hierarchical data structures. There exist various classifications of PAM [Gaed97], [Same90b]. These classifications are necessarily ambiguous, as many PAM are hybrid and cannot be solely arranged to one group. [Gaed97] report the following categorization for point access methods:

- Multidimensional hashing access methods. These methods use 1-dimensional hashing to index d-dimensional points. Although there is no total ordering of d-dimensional objects in one dimension, these methods use heuristic techniques to ensure that two objects that are close to each other in the multidimensional space, will be indexed the same, or close buckets. Examples of such hashing methods are the grid file[Niev81] and EXCELL[Tamm82].
- Hierarchical access methods. These methods use hierarchical data structures to manage point data. PAM that fall in this category are the Quadtree[Fink-Bent74], the k-d-tree[Bent75], and k-d-B-tree[Robi81]. Access methods such as the Buddy tree[Seeg90] and the BANG file[Free87] can be considered as hybrid, since they incorporate techniques of both hierarchical and hashing methods.

Multidimensional access methods often make use of the so called *space-filling curves* to preserve spatial proximity when ordering multidimensional points in the one-dimensional space. These techniques suggest a total ordering of spatial objects, ensuring with a high probability that if two objects are located close together in the original space then they will be close together in the total order. [Same90b] provides a good overview of space-filling curves for point data.

### *2.4.2 Spatial Access Methods (SAM)*

Point access methods cannot directly be used to manage objects with a spatial extent. Spatial access methods are often extensions of PAM, used to cover this need. [Gaed97] classify these methods according to the techniques they use to extend PAM, as follows:

- Object mapping methods. These methods map geometric objects into points in a higher-dimensional space. For instance, a rectangle in $R^2$ can be viewed as a point in $R^4$. They then use existing PAM to manage the points. One alternative approach used by such methods is the decomposition of geometric objects into simple ones (e.g. rectangles) and ordering of the simple objects using space filling curves.
- Object bounding methods (overlapping regions). Being the most popular SAM, these methods decompose the space in a hierarchical manner. Objects are stored at the leaves of the hierarchical structures, and intermediate nodes facilitate efficiency at searching. Nodes at the same level may overlap each other, so the number of paths that have to be followed in search of an object can vary. The most promising object bounding methods are the R-tree[Gutt84] and R*-tree[Beck90].
- Clipping methods. These methods use hierarchical data structures, as the object bounding methods do, but they use clipping of objects to prevent overlapping of intermediate nodes at the same level. In this way they ensure that only one path of the hierarchical structure will be traversed in search of an object. Objects are clipped, and stored in several nodes, in order to guarantee this non-overlapping feature. Example of such access methods is the R+-tree[Sell87]
- Multiple layers. Multiple layers methods partition the space more than one time and each partition is referred to as a layer. Layers are organized in an hierarchical manner, and partition regions within the same layer do not overlap. Each layer may use a different algorithm to partition the space. An object is stored to a region of the lowest possible layer in the hierarchy that can store the object without clipping it. A characteristic multiple layer access method is the Multi-layer grid file [Six88].

## 3. Spatial Data Structures

In this section we make a brief description of some of the most important point and spatial access methods. It is impossible to include descriptions of all access methods that have been proposed, within the

limitation space of this work, however, we feel that the few methods presented below will give the reader an idea about the multidimensional access methods' functionality.

## 3.1 Hashing access methods

### *3.1.1 The Grid file and its variants*

As a typical representative for an access method based on hashing, we will first discuss is the *grid file* and some of its variants [Hinr85, Ouks85, Whan85, Sixw88, Blan90]. The grid file is a variation of the grid method, which relaxes the requirement that cell division lines should be equidistant. Its goal is to retrieve records by at most two disk accesses and to efficiently handle range queries. This is done by using a grid directory consisting of grid blocks that are analogous to the cells of the fixed-grid method. All records in one grid block are stored in the same bucket. However, several grid blocks can share a bucket as long as the union of these grid blocks forms a k-dimensional rectangle in the space of records. Although the regions of the buckets are piecewise disjoint, together they span the space of records. To guarantee that data items are always found with no more than two disk accesses for exact match queries, the grid itself is kept in main memory, represented by *d* one-dimensional arrays called *scales.*

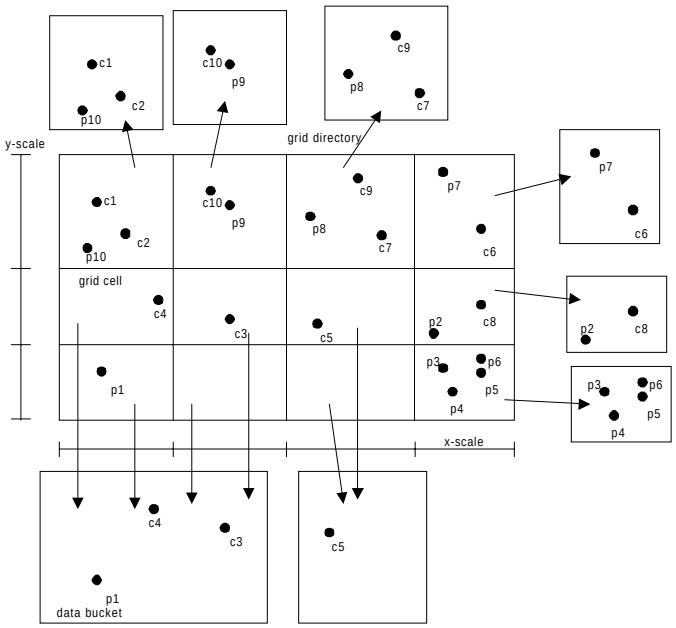


Figure 2: Grid File

Figure 2 shows a grid file which has bucket capacity of four data points. The center of the figure shows the directory with scales on the *x*- and *y*-axes. To answer an exact match query, one first uses the scales to locate the cell containing the search point. If the appropriate grid cell is not in the main memory, one disk access is necessary, and the loaded cell contains a reference to the page where to find possibly match data.

Merrett and Otoo describe a technique termed *multipaging* [Merr78, Merr82], which is similar to the grid file. It also uses a directory in the form of linear scales called *axial arrays*. Instead of using a grid directory, however, multipaging accesses a data page and its potential overflow chain using an address computed directly from the linear scales. There are two variants of multipaging. In dynamic multipaging [Merr82], performance is controlled by setting a bound on the probe factor (defined as the average number of pages accessed in a probe). In static multipaging [Merr78], performance is controlled by setting a bound in the load factor, or the average page occupancy.

Comparing the grid file and multipaging, we find that the grid file uses multipaging as an index to the grid directory. Therefore multipaging saves space without requiring a grid directory, but this is at a cost of requiring bucket overflow areas. This means that multipaging can obtain good average-case performance,

but it cannot guarantee record retrieval with two disk accesses. In addition, insertion and deletion in multipaging involves whole rows or columns (in the two dimensional case) of data pages when splitting or merging buckets, while the grid file can split one page at a time and localize more global operations in the grid directory.

### *3.1.2 Other hashing methods*

Closely related to the grid file is the EXCELL method (Extendible CELL) proposed by Tamminen [Tamm82]. It is a bintree together with a directory in the form of an array providing access by address computation. It can also be viewed as an adaptation of extendible hashing [Fagi79] to multidimensional point data. In contrast to the grid file, where the partitioning hyperplanes may be spaced arbitrarily, the EXCELL method decomposes the universe *regularly*; all grid cells are of equal size. In order to maintain this property in the presence of insertions, each new split results in the having of all cells and therefore in the doubling of the directory size. To alleviate this problem, Tamminen [Tamm83] later suggested a hierarchical method, similar to the multilevel grid file of Whang and Krishnamurthy [Whan85]. Overflow pages are introduced to limit the depth of the hierarchy.

Another hashing method we consider here is the *two-level grid file*. The basic idea of it is to use a second grid file to manage the grid directory. The first of the two levels is called the *root directory*, which is a coarsened version of the second level, the actual grid directory. Entries of the root directory contains pointers to the directory pages of the lower level, which in turn contain pointers to the data pages. By having a second level, splits are often confined to the subdirectory regions without affecting too much of their surroundings.

The twin grid file is the other hashing method which tries to increase space utilization compared to the original grid file by introducing a second grid file. The relationship between these two grid files is not hierarchical but somewhat more balanced. Both grid files span the whole universe. The distribution of the data among the two files is performed dynamically. If the number of points in a bucket exceeds the given limit, the twin grid file tried to redistribute the points among the two grid files. A transfer of points from the primary file *P* to the secondary file *S* may lead to a bucket overflow in *S*. It may, however, also imply a bucket underflow in *P*, which may in turn lead to a bucket merge and therefore to a reduction of buckets in *P*. The overall objective of the reshuffling is to minimize the total number of buckets in the two grid files *P* and *S*.

## 3.2 Quadtrees

Quadtrees are one of the first data structures for higher-dimensional data. They were developed by Finkel and Bentley in 1974[Fink-Bent74]. Since then, there have been been hundreds of papers dealing with quadtrees. The surveys and two books by Samet[Same84, Same88, Same90a Same90b] give an extensive overview of the various types of quadtrees and their applications.

A quadtree is a rooted tree in which every internal node has four children. Every node in the quadtree corresponds to a square. If a node *v* has children, then their corresponding squares are the four quadrants of the square of *v* — hence the name of the tree. This implies that the squares of the leaves together form a subdivision of the square of the root. We call this subdivision the *quadtree subdivision*. Figure 3 gives an example of a quadtree and the corresponding subdivision. The children of the root are labeled NE, NW, SW, and SE to indicate to which quadrant they correspond; NE stands for the north-east quadrant, NW for the north-west quadrant, and so on.

The recursive definition of a quadtree immediately translates into a recursive algorithm: split the current square into four quadrants partition the input data set accordingly, and recursively construct quadtrees for each quadrant with its associated input data set.

Quadtrees can be differentiated on the following bases:

a) The type of data they are used to represent
b) The principle guiding the decomposition process
c) The resolution (variable or not)

Currently they are used for point data, areas, curves, surfaces, and volumes. The decomposition may be into equal parts on each level (i.e., regular polygons and termed a regular decomposition), or it may be governed by the input. In computer graphics this distinction is often phrased in terms of image-space hierarchies versus object-space hierarchies, respectively [Suth 74]. The resolution of the decomposition (i.e., the number of times that the decomposition process is applied) may be fixed beforehand, or it may be governed by properties of the input data. For some applications we can also differentiate the data structures on the basis of whether they specify the boundaries of regions (e.g., curves and surfaces) or organize their interiors (e.g., areas and volumes).

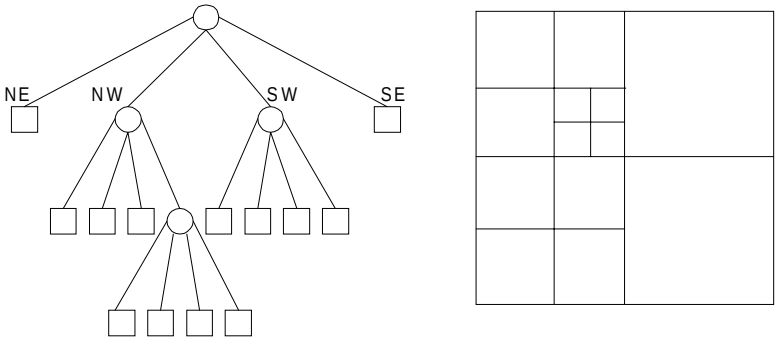


Figure 3: A quadtree and the corresponding subdivision

### *3.2.1 Variants of Quadtrees*

The first example of a quadtree representation of data is concerned with the representation of two-dimensional binary region data. The most studied quadtree approach to region representation, called a region quadtree is based on the successive subdivision of a bounded image array into four-equal sized quadrants. If the array does not consist entirely of 1s or entirely 0s (i.e., the region does not cover the entire array), then it is subdivided into quadrants, subquadrants, and so on, until blocks are obtained that consists entirely of 1s or entirely of 0s; that is, each block is entirely contained in the region or entirely disjoint from it. The region quadtree can be characterized as a variable resolution data structure.

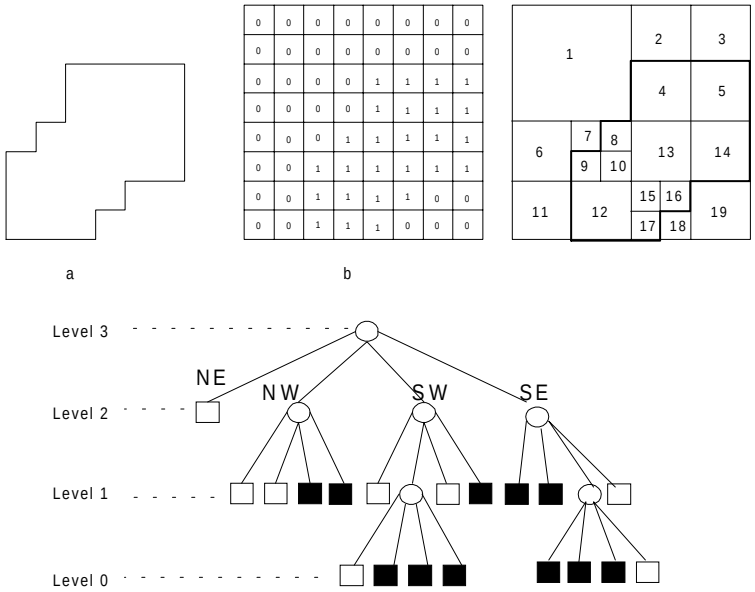


Figure 4: An example region, its binary array, its maximal blocks (block in the region are shaded), and the corresponding quadtree

As an example of the region quadtree, consider the region shown in Figure 4a represented by the $2^3 2^3$ binary array in Figure 4b.

Quadtree-like data structures can also be used to represent images in three dimensions and higher. The octree [Hunt78, Meag82] data structure is the three-dimensional analog of the quadtree. It is constructed in the following manner. We start with an image in the form of a cubical volume and recursively subdivide it into eight congruent disjoint cubes (calles octants) until blocks are obtained of a uniform color or a predetermined level of decomposition is reached.

Multidimensional point data can be represented in a variety of ways. The representation ultimately chosen for a specific task is influenced by the type of operations to be performed on the data. In higher dimensions (i.e., greater than 3) it is preferable to use the k-d tree [Bent75] as every node has degree 2 since the partitions cycle through the different attributes. PR quadtree [Oren82, Same90b]is based on a regular decomposition. The PR quadtree is organized in the same way as the region quadtree. The difference is that leaf nodes are either empty (i.e., white) or contain a data point (i.e., black) and its coordinates. A quadrant contains, at most, one data point. For example, Figure 5 is the PR quadtree corresponding to the point quadtree of figure 6 (for more details, see Section 2.6.2 of [Same90b]).

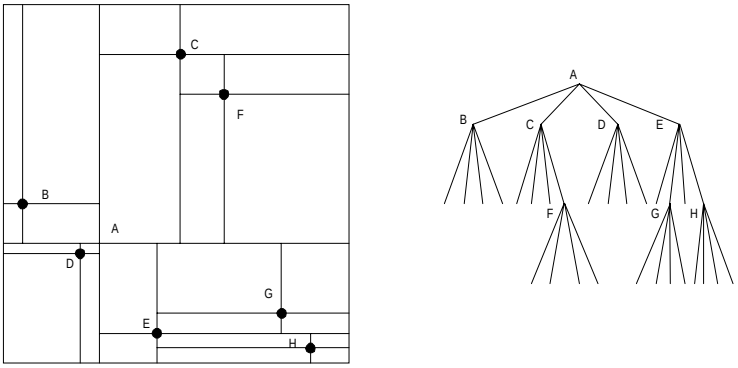


Figure 5**:** A point quadtree and the records it represents

The PR quadtree representation can also be adapted to represent a region that consists of a collection of polygons (termed a *polygonal map*). The result is a family of representations referred to collectively as a PM quadtree [Same85]. The PM quadtree family represents regions by specifying their boundaries; this is in contrast to the region quadtree, which is based on a description of the region's interior.

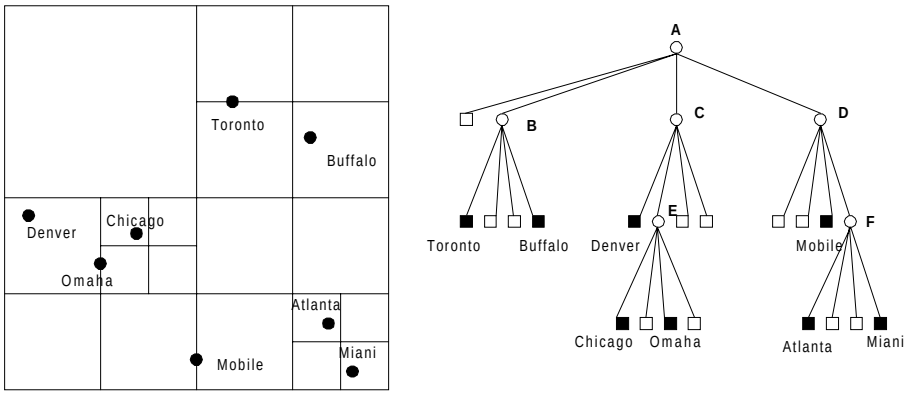


Figure 6 A PR quadtree and the records it represents

As an example of the PM quadtree family, consider the $PM_1$ quadtree. The polygonal map is repeatedly subdivided into four equal-sized quadrants until we obtain blocks that do not contain more than one line. To deal with lines that intersect other lines, we say that if a block contains an endpoint *p* of a line, we permit it to contain more than one line provided that *p* is an endpoint of each of the lines it contains. A

block can never contain more than one endpoint. For example, Figure 7 is the block decomposition of the $PM_1$ quadtree and its tree representation corresponding to the polygonal map.

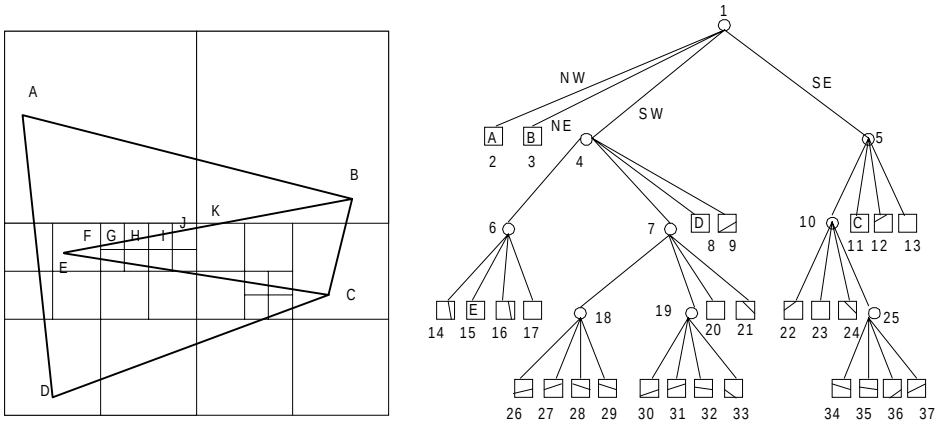


Figure 7 $PM_1$ quadtree and corresponding Tree representation

The **$PM_1$** quadtree has also been adapted to three dimensional images. We term the result a **PM** octree. The decomposition criteria are such that no node contains more than one face, edge, or vertex unless all the faces meet at the same vertex or are adjacent to the same edge. For example, Figure 8b is **PM** octree decomposition of the object in Figure 8a. This representation is quite useful since its space requirements for polyhedral objects are significantly smaller than those of a region octree. (For more details, see Section 5.3 of [Same90b].)

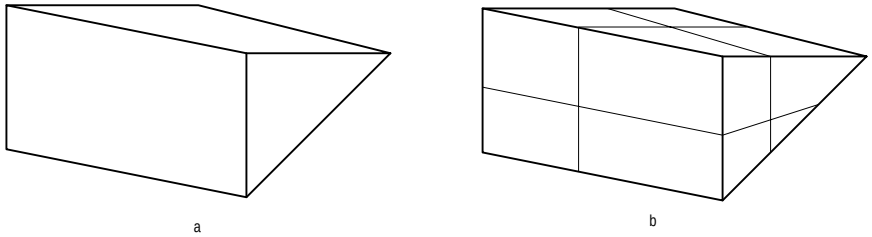


Figure 8 Example three-dimensional object and its corresponding PM octree

## 3.3 The k-d-tree and its variants

One of the most prominent multidimensional data structure is the *k-d-tree*[Bent75]; a binary search tree that stores points of the k-dimensional space. At each intermediate node, the k-d-tree divides the k-dimensional space in two parts by a (k-1)-dimensional hyperplane. The direction of the hyperplane, i.e. the dimension based on which the division is made, alternates between the k possibilities from one tree level to the next. Each splitting hyperplane contains at least one point, which is used as the hyperplane's representation in the tree.

Figure 9 illustrates a 2-d-tree with some point data in it. Note that we compare x-coordinate values on the even depths of the tree (the root is considered to be at depth 0), and y-coordinate values at the odd depths. If a node P has a n-discriminator, then all nodes having their n-coordinate value less than that of P are located under P's left son, and all nodes with a n-coordinate value greater than or equal to that of P are located under P's right son.

Searching and insertion of new nodes are straightforward. Deletion may cause re-organization of the tree under the deleted node, thus it can be more complicated.

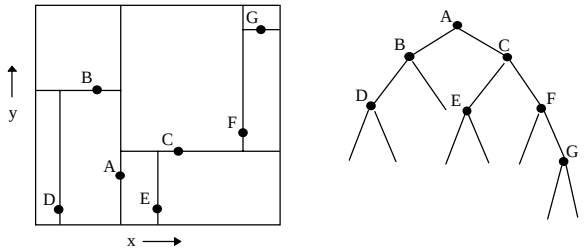


Figure 9: A distribution of points in the plane and the correspondent 2-d-tree

The tree's structure depends heavily on the insertion order of the points. Another disadvantage of the k-d-tree is that as the division hyperplanes are defined by the position of the points, they do not divide the plane at the best possible positions, resulting in an unbalanced tree. An improved version proposed in [Frie77] is the *adaptive k-d-tree*. When splitting, the adaptive k-d-tree chooses a hyperplane that divides the space in two sub-spaces with equal number of points. The hyperplanes are still parallel to axes, but they do not contain a point, and they do not have to strictly alternate. Interior nodes of the tree contain the dimension (e.g. x or y), and the coordinate of the respective split. All points are stored in the leaves, and a leave is can contain up to a fixed number of points, if this number is exceeded, a split takes place. Intuitively, the k-d-tree is a rather static structure; balance maintenance is difficult if frequent insertions and deletions occur. It works well when the data is known a-priori, and there are rare updates in the tree.

One thing to be noted for the above structures is that they are *main memory data structures*. That is, they do not account for paged secondary memory, and are therefore not suitable for large spatial databases. The *k-d-B-tree* [Robi81] combines properties of both the adaptive k-d-tree and the B-tree [Come79] to face this weakness. It again uses hyperplanes to divide the space; this time arbitrarily more than one hyperplanes divide a tree node (depending on the tree's storage utilization) in a corresponding number of disjoint regions. All nodes of the tree correspond to disk pages. A leaf node stores the data points that are located in the respective partition the leaf defines. Like the B-tree, the k-d-B-tree is perfectly balanced, however, it cannot ensure storage utilization.

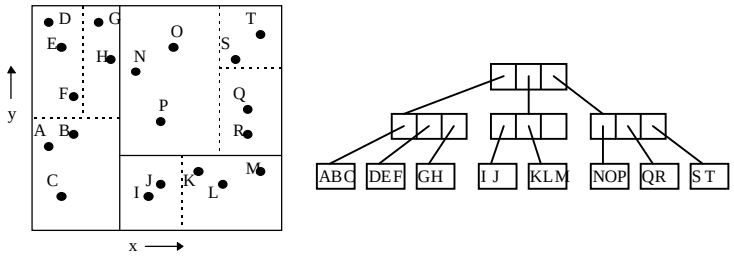


Figure 10: A distribution of points in the plane and the correspondent k-d-B-tree

Figure 10 shows how a point distribution can be stored in a k-d-B-tree. The solid lines correspond to the hyperplanes of the root-level, whereas the dotted lines represent the next level's hyperplanes.

## 3.4 R-trees

The R-trees are hierarchical data structures, meant for efficient indexing of multidimensional objects with spatial extent. R-trees are used to store, instead of the original space objects, their *minimum boundary boxes* (MBBs). The MBB of a n-dimensional object is defined to be the minimum n-dimensional rectangle that contains the original object. Similar to B-trees, the R-trees are balanced and they ensure efficient storage utilization.

The R-trees manage MBBs and not real objects, thus they cannot fully answer a query, unless the objects in the database are equal to their MBBs. In general, they are used to efficiently solve the *filter* step of a query, that is finding the database objects whose MBB intersects with the MBB of the query object.

### *3.4.1 The R-Tree*

The R-tree[Gutt84] is the father of all R-tree variants. Each R-tree node corresponds to a disk page and a n-dimensional rectangle. Each non-leaf node contains entries of the form (*ref, rect*), where *ref* is the address of a child node and *rect* is the MBB of all entries in that child node. Leaves contain entries of the same format, where *ref* points to a database object, and *rect* is the MBB of that object.

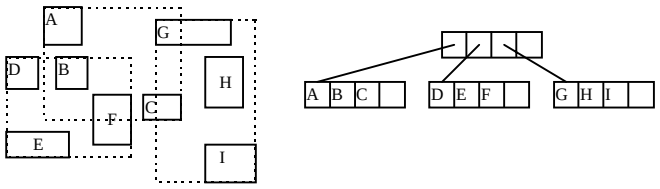


Figure 11: An R-tree for a set of 2-d rectangles

The rest of the R-tree properties include:

- Let M be the number of entries that can fit in a node, and let m be the minimum number of entries per node, $2 \leq m \leq \lceil M/2 \rceil$. Every node contains between m and M nodes, unless it is the root. If the number of entries in a node falls under the m bound after an entry deletion, the node is deleted, and the rest of its entries are distributed among the sibling nodes.
- The root contains at least 2 entries, unless it is a leaf.
- The tree is height-balanced; every leaf node has the same distance from the root. The height of the tree is at most $\lceil \log_m N \rceil$ for n index records (N > 1).

Figure 11 illustrates a set of 2-d objects in the plane, stored in an R-tree. The dotted rectangles are the MBBs of the root entries, and the solid rectangles are the MBBs of the objects stored at leaves. Note that the MBBs of entries at the same node may intersect one another.

Searching in an R-tree is done in a similar way as in a B-tree. For both point and region queries, the paths where *rect* intersects with the query object are followed. In contrast to the B-tree, the R-tree does not guarantee that traversing one path of the tree is enough when searching for an object, as the MBBs of entries in the same nodes may overlap one another. In the worst case, the search algorithm may have to visit all index pages, in order to answer a query.

Inserting an object to the R-tree, includes inserting its MBB to the R-tree along with a reference of the object to the *ref* field of the new entry. Only one path of the tree is traversed and the new entry is inserted to a leaf node. If the MBB of the object intersects many entries of an intermediate node, we follow the child whose MBB is less enlarged after the insertion. In case of a tie, we apply other criteria, such as the node's cardinality, or MBB area size. The object is inserted only at one leaf and if it causes the leaf page to overflow, we split the page in two, again after applying several splitting criteria. The split can be propagated to the ancestor nodes. If an insertion causes enlargement of the leaf page's MBB, we adjust it properly and propagate the change upwards.

Deletion in an R-tree requires an exact match query for the object, at first. If the object is found in a leaf, it is deleted. Again the deletion may cause a modification in the tree's structure, as it can cause the leaf page where from it is deleted, to underflow (the number of entries may fall under m). In the case of an underflow, the whole node is deleted, and all its entries are stored in a temporary buffer, and reinserted in the tree. As for insertion, deletion may affect the MBB of the page. In that case, we propagate the change up along the search path.

Minimizing the overlap between sibling nodes is an important issue, concerning the searching performance in an R-tree [Rous85]. Guttman [Gutt84] suggests various policies to minimize overlap during insertion. Roussopoulos et al. [Rous85] introduce a *packing* technique, which builds optimal R-tree, provided all inserted data is known a-priori. The variant of R-tree that is considered to best handle dynamic object insertion is the $R^*$-tree[Beck90], discussed later.

### *3.4.2 The R+-Tree*

The $R^+$-tree was introduced by Sellis et al.[Sell87] as a way to overcome the problem of inefficient searching that arises when sibling nodes overlap in the R-tree. As a direct solution to these problems they use *clipping*, i.e. there is no overlap between intermediate nodes of the tree at the same level, and objects that intersect more than one MBB at a specific level are clipped and stored on several different pages. As a result, point queries on the $R^+$-tree require traversing only one path of the tree. The price to pay is the increase of storage requirement of the tree. Figure 12 presents a $R^+$-tree for the rectangles of figure 11. Rectangles G and F are clipped and stored in twice in the tree.

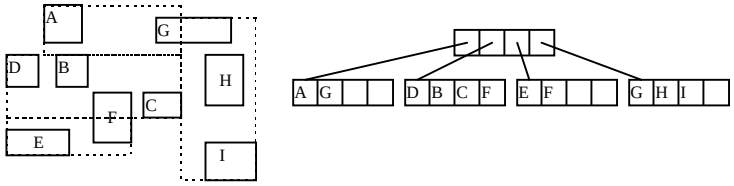


Figure 12: An $R^+$-tree for a set of 2-d rectangles

Insertion requires following multiple paths of the tree, since the inserted object may intersect more than one intermediate node, and its clipping parts should be inserted in leaves under all such nodes. As the insertion of an object's part may enlarge the MBB of the page, the insertion algorithm has to prevent possible overlapping between sibling pages. In some cases overlapping is inevitable, and we should consider removing and re-inserting several objects to properly reorganize the structure of the tree. Node splits are done in a similar way as in R-trees case. One important difference is that here a split may propagate to the children of the node, apart from its parent. This, because a split to a parent node may introduce a space partition that affect the children nodes.

Object deletion is succeeded by first finding the pages that contain fragments of the object and then removing the fragments. If underflow occurs, we try to merge the node with its siblings. Sometimes this is not possible without losing the disjoining property of the tree, therefore the $R^+$-tree cannot guarantee a minimum storage utilization.

### *3.4.3 The R*-Tree*

Several weaknesses of the original R-tree insertion algorithms stimulated Beckmann et al. [Beck90] to work on an improved version of the R-tree, the $R^*$-tree. This version introduces a new insertion policy, that crucially improves the performance of the tree. The main objective of this policy is to minimize the overlap region between sibling nodes in the tree. A straightforward advantage of this is the minimization of the tree paths that are traversed at an object search. The advantages of the new insertion algorithm over its R-tree respective can be summarized as follows:

- While traversing the insertion path, the insertion algorithm follows the nodes, whose MBB has the minimum increase of overlap. Thus, the search performance is improved [Rous85].
- Whenever a new entry has to be stored into a full node, the node is not necessarily split, but some entries are deleted, and re-inserted to sibling nodes. The entries for re-insertion are chosen to be those with maximum distance from the center of the node's MBB. This feature of the algorithm increases storage utilization, and improves the quality of the partition, making it almost independent of the sequence of insertions.
- The algorithm for splitting a node is totally different from its R-tree equivalent. First, the algorithm decides the axis with respect to which the split will take place. Then, the projections of the MBBs over the split-axis are sorted according to the value of their left end point. This sequence can be divided to two sub-sequences, in M-2m+1 ways. Among these splits, the algorithm chooses the one that results in a minimum overlap between the MBBs. This algorithm is proved to achieve better quality of the MBBs partition over the tree.

### *3.4.4 Using R-trees to process spatial joins*

In section 2.3.2 we briefly described a multi-step technique for processing spatial joins proposed in [Brin94]. In this section we will focus on the first step of the technique, that is computing the MBR-Join of two spatial relations.

Brinkoff et al. in [Brin93] extensively analyzed the problem of MBR-spatial join and developed a series of techniques that efficiently solve it. For their techniques they have made the assumption that the spatial relations to be joined are indexed by a pair R,S of R*-trees. The first join approach, that takes advantage of the R*-tree structure, is based on the following fact: if two directory entries $E_R \in R$ and $E_S \in S$ do not intersect, there will be no pair ($rect_R$, $rect_S$) of rectangles that intersect, where $rect_R$, $rect_S$ are under the sub-trees of $E_R$ and $E_S$, respectively. The corresponding algorithm for this approach is illustrated in figure 13. This algorithm assumes that both trees have the same depth. It can be easily extended to the general case by observing that if we reach the leaf-level of one tree, the rectangle comparison at this level can be replaced by a range query to the subtrees of the other tree at the same level.

```
SpatialJoin1 (R,S: R_Node)
    FOR (all E_S ∈ S) DO
        FOR (all E_R ∈ R with E_R.rect ∩ E_S.rect ≠ ∅) DO
            IF (R is a leaf page) THEN (*S is also a leaf page*)
                output (E_R, E_S)
            ELSE
                ReadPage(E_R.ref); ReadPage(E_S.ref);
                SpatialJoin1(E_R.ref, E_S.ref);
```

Figure 13: A simple MBR-join algorithm using R*-trees

The above algorithm implements exhaustive comparisons between all possible pairs of the entries at the same level. It can be improved by using two techniques; restricting the search space, and applying a plane sweep algorithm that improves the computation cost of calculating which pairs from two sets of rectangles intersect. The use of both these techniques is illustrated in figure 14. The first technique prunes the search space by considering only those rectangles that intersect the intersection of the MBRs of the pages where they are contained; if a rectangle does not intersect this area, there is no way it will be in any answer pair at this level. The second technique sweeps a line across the x-axis, by adding/removing rectangles whose x-projection intersects with the sweep line. Each time 2 x-projections of rectangles are found to intersect each other, their corresponding y-projections are tested and if they intersect too, the pair is reported to the output set.

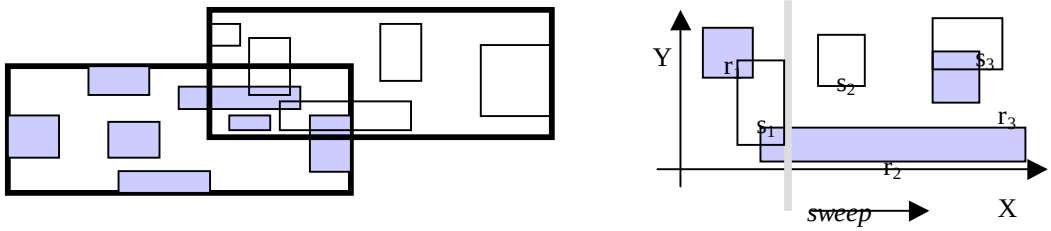


Figure 14: CPU-time tuning: (a) restricting the search space (b) plane sweep algorithm

Another improvement of the MBR-join algorithm concerns finding the best sequence of pairs of pages required for computing the join at the next level of the R*-trees. This order is important as far as I/O-time tuning is concerned; we are interested to access one disk page as few times as possible, in order to minimize the I/O time of the algorithm. Hence, when we have a set of pairs of pages to be processed, we apply a "pinning" technique, that is we count how many times each page is contained in a pair and we consider processing first those pairs, which contain the page which is contained in most pairs. By using this technique, a frequently used page will remain "pinned" in main memory, until it needs no further processing.

# 4. Comparative studies of multidimensional access methods

We have already examined various multidimensional access methods. In this section we try to compare them with their performance and make some observations. The performance of a particular data structure depends on many factors and parameters. For example, a spatial access method that performs reasonably well for data rectangles, may fail for data line segments.

There are factors of unpredictable impact, including the hardware used, the settings of the operating system, buffer size, page size and the data sets. The parameters that affect the performance include if the data distribution is non-uniform or not, if there is a suitable modeling or not to show the behavior of spatial access method, the amount of data, the density in the data space, and the degree of clustering. Furthermore, the performance is usually measured in terms of number of disk accesses, the search time, the deletion time, etc.

There were comments by researchers that at present no access method has proven itself to be much superior to all its methods. Although if there is one experimental result declares one structure as the definite winner, another experimental result may prove the same structure as inferior. The reasons why this makes such comparisons so difficult, it is because there are many different criteria to define optimality, and so many parameters that determine performance.

As a result, the following experimental results in tabular form will just show you an overview of performance done by researchers on a variety of spatial data structures. In addition, the following multidimensional access methods were commented to be among the best performing ones in general, without any ranking presented.

- Buddy (hash) tree [Seeg90]
- Cell tree with oversize shelves [Gunt97]
- Hilbert R-tree [Kame94]
- KD2B-tree [Oost90]
- PMR-quadtree [Nels87]
- $R^+$-tree [Sell87]
- R*-tree [Beck90]

The performance results are shown as follows in tabular form for your easy reference :

**Legend**: ">" means "the performance is better than"
"=" means "the performance is about the same"

| **Reference** | **Result/Observations** | **Conditions** |
|---|---|---|
| [Pelo94] | ($R^+$-tree, a quadtree variant) > R*-tree | Used polygons rather than line segments as test data. Based on point queries. Physical clustering must be provided, otherwise, reading a single index page may induce several page faults. |
| [Gree89] | $R^+$-tree > R-tree > k-d-B-tree | When less overlap between data rectangles |
| [Beck90] | $R^*$-tree > Variants of the R-tree<br>$R^*$-tree has best storage utilization and insertion times. | For all data list and queries, only number of disk accesses is measured. |
| [Kame94] | Hilbert R-tree slightly better than $R^*$-tree<br><br>Hilbert R-tree has better search result, while updates take about the same as for | |

| | | |
|---|---|---|
| | the $R^*$-tree.<br>Hilbert codes can therefore be used for bulk insertion into dynamic $R^*$-tree. | |
| [Ooi90] | skd-tree > R-tree<br>skd-tree requires more space than R-tree.<br><br>skd-tree > extended k-d-tree with overflow pages | For large page size, the performance is in term of number of page accesses per search operation. |
| [Gunt91] | Cell tree > R-tree and $R^+$-tree<br><br>Cell tree requires up to 2 times more space. | The average page accesses per search is less. |
| [Gaed97] | Cell tree with oversize shelves > $R^*$-tree, and hB-tree | Oversize shelves lead to great improvements for access methods clipping. |
| [Oost90] | KD2B-tree (a variant of k-d-tree) > R-tree | When compared with the query times. |
| [Hoel92] | PMR-quadtree = $R^*$-tree = $R^+$-tree<br><br>$R^+$-tree shows the best insertion performance<br>$R^*$-tree occupies the least space and is more compact in term of the data structure itself. | When use line segments as test data for indexing. |
| [Krie90] | (Buddy tree, BANG file) > (hB-tree, 2-level grid file)<br><br>(Buddy tree, BANG file) > R-tree | For cluster data & a query range of size 10% of the data, no performance different between buddy and BANG.<br>If the query range drops to about size 0.1%, buddy performs about twice as fast.<br><br>For all data distributions in terms of measuring the number of page accesses. |
| [Seeg91] | Buddy tree with transformation > R-tree<br><br>Supports fast insertions with low storage utilization.<br><br>Buddy tree with overlapping regions > Buddy tree with transformation > $R^*$-tree | Techniques (Clipping [object duplication], Overlapping regions [object bounding], & Transformation [object mapping]) implemented on top of buddy tree.<br>For queries on intersection & containment, but NOT for large query regions. |
| [Seeg91] | Buddy tree with clipping is better | When the data set contains uniformly distributed rectangles of varying size.<br>But it failed completely for certain distributions, since they produced unmanageably large files. |
| [Hutf90] | R-file > R-tree | |

| | | |
|---|---|---|
| | R-file has a 10-20% performance advantage over the R-tree on a data set with a high degree of overlap. | |
| [Hutf88] | R-tree with splitting strategy > R-tree<br><br>R-tree with an improved variant of z-hashing needs very less seek operations than R-tree, and the average storage utilization is higher. | |
| [Smit90] | Compare the performance on insertion, deletion & search operation for the zkdB$^+$-tree, grid file, the R-tree, and the R$^*$-tree.<br><br>R and R$^+$-tree are fairly good on insertions & deletions, but superior in search operation.<br><br>R$^+$-tree is not good for general purpose applications, due to its poor space utilization. | |

Table 1: Comparative studies reported by various researchers

# 5. Conclusions

In this paper we focused on giving the reader a general overview of the various multidimensional access methods developed during the past two decades. From the variety of data structures and their experimental performances, we can have ideas on their pros and cons. But still this survey did not try to resolve which spatial access methods are more reliable and superior than the others.

When these methods come to be used in applications, vendors of commercial products usually select access methods that are easy to understand and implement. Performance seems to be a good reference for the selection. This is quite understandable, because vendors try to take a structure that is simple and robust, and to optimize its performance by a highly tuned implementation. One way to optimize their performance is to implement several access methods and a code optimizer. The optimizer applies the appropriated access method to answer to different type of queries.

In addition, the performance results of access methods are essential as they often discover deficiencies and problems that are not obvious from a theoretical model. Future work can be extended to set up a standardized testbed for benchmarking and comparing access methods under different conditions. Also it is essential to provide platform-independent access to the implementations of a board variety of access methods. Until then, most performance benchmarks will have significant effects in determining which access method can provide the best fit for different type of applications.

# References

[Beck90] Beckmann, N., H.-P. Kriegel, R. Schneider, and B. Seeger, "The R$^*$-tree: An Efficient and Robust Access Method for Points and Rectangles," In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 322-331, 1990

[Bent75] Bentley, J. L., "Multidimensional Binary Search Trees Used For Associative Searching," *Communications of the ACM*, 18(9), 509-517, 1975

[Brin94] Brinkhoff, T., H-P. Kriegel, R. Schneider, and B. Seeger, "Multi-Step Processing of Spatial Joins," In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 197-208, 1994

[Brin93] Brinkhoff, T., H-P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins using R-trees," In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 237-246, 1993

[Bent75] Bentley, J. L., "Multidimensional Binary Search Trees Used For Associative Searching," *Communications of the ACM*, 18(9), pp. 509-517, 1975

[Blan90] Blanken, H., A. Ijbema, P. Meek, and B van den Akker, "The generalized grid file: Description and performance aspects," In *Proceeding of 6th IEEE International Conference on Data Engineering*, pp. 380-388, 1990

[Come79] Comer, D., "The ubiquitous B-tree," *ACM Computing Surveys*, 11(2), pp. 121-138, 1979

[Fagi79] Fagin, R., J. Nievergelt, N. Pippenger, and R. Strong, "Extendible hashing: A fast access method for dynamic files," ACM Transaction on Database Systems, 4(3), pp. 315-344

[Fink74] Finkel, R.A. and J. L. Bentley, "Quad trees: a data structure for retrieval on composite keys," *Acta Inform.*, 4:11-9, 1974

[Free87] Freeston, M., "The BANG file: A new kind of grid file," In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 260-269, 1987

[Frie77] Friedman, J. H., J. L. Bentley, and R. A. Finkel, "An Algorithm For Finding Best Matches in Logarithmic Expected Time," *ACM Transactions on Mathematical Software*, 3(3), pp. 209-226, 1977

[Gaed97] Gaede, V. and O. Günther, "Survey on Multidimensional Access Method," *Technical Report ISS-16*, Department of Economics and Business Administration, Humboldt University Berlin, revised version, 1997.

[Günt91] Gunther, O. and J. Bilmes (1991) "Tree-based access methods for spatial databases: Implementation and performance evaluation." IEEE Trans. Knowledge and Data Eng. 3(3), 342-356.

[Günt93] Günther, O., "Efficient Computations of Spatial Joins," In *Proc. 9th International Conference on of Data Engineering*, pp. 50-59, 1993

[Gutt84] Guttman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 47-57, 1984

[Hoel92] Hoel, E.G. and H. Samet (1992) "A qualitative comparison study of data structures for large segment databases." In Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 205-214.

[Hinr85] Hinrichs, K., "The grid file system: implementation and case studies of applications," PhD. dissertation, Institute fur Informatik, ETH, Zurich, Switzerland, 1985

[Hunt78] Hunter, G. M., "Efficient computation and data structures for graphics," PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ.

[Hutf88] Hutflesz, A., H.W. Six, and P. Widmayer (1988a) "Globally order preserving multidimensional linear hashing." In Proc. 4th IEEE Int. Conf. on Data Eng., pp.572-579.

[Hutf90] Hutflesz, A., H.W. Six, and P. Widmayer (1990) "The R-file: An efficient access structure for proximity queries." In Proc. 6th IEEE Int. Conf. on Data Eng., pp.372-379.

[Kame94] Kamel, I. and C. Faloustsos (1994) "Hilbert R-tree: An improved R-tree using fractals." In Proc. 20th Int. Conf. On Very Large Data Bases, pp. 500-509.

[Krie90] Kriegel, H.P., M. Schiwietz, R. Schneider, and B.Seeger (1990) "Performance comparison of point and spatial access methods." In A. Buchmann, O. Gunther, T.R. Smith, and Y.F. Wang, "Design

and Implementation of Large Spatial Database Systems", Number 409 in LNCS, Berlin/Heidelberg/New York, pp. 89-114.

[Meag82] Meagher, D., "Geometric modeling using octree encoding,|" *Computer Graphics and Image Processing*, 19, 2 (June), pp. 129-147

[Merr78] Merrett, T. H., "Multidimensional paging for efficient database querying," *Proceedings of the International Conference in Management of Data*, Milan, June 1978, pp. 277-289

[Merr82] Merrett, T. H. and E. J. Otoo, "Dynamic multipaging: a storage structure for large shared data banks," in *Improving Database Usability and Responsiveness*, P. Sheuermann, ed., Academic Press, New York, 1982, pp. 237-254

[Niev81] Nievergelt, J., H. Hinterberger, and K. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," In A. Duijvestijn and p. Lockemann (Eds.), *Proc. 3rd ECI Conf.*, Number 123 in LNCS, Berlin/Heidelberg/New York, pp. 236-251, Springer-Verlag, 1981

[Ooi90] Ooi, B.C. (1990) "Efficient Query Processing in Geographic Information Systems." Number 471 in LNCS. Berlin/Heidelberg/New York.

[Oost90] Oosterom, P. (1990) "Reactive Data Structures for GIS" Ph.D. thesis, University of Leiden, The Netherlands.

[Oren82] Orenstein, J. A., "Multidimensional tries used for associative searching," Information Processing Letters 14, 4(June 1982), pp. 150-157

[Ouks85] Ouksel, M., "The interpolation-based grid file," *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Portland, OR.

[Pelo94] Peloux, J.; G. Reynal and M. Scholl (1994) "Evaluation of spatial indices implemented with the $O_2$ DBMS.

[Robi81] Robinson, J. T., "The K-D-B-tree: A Search Structure For Large Multidimensional Dynamic Indexes," In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 10-18, 1981

[Rous85] Roussopoulos, N. and D. Leifker, "Direct Spatial Search on Pictorial Databases Using Packed R-Trees," In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 17-31, 1985

[Same84] Samet, H., "The quadtree and related hierarchical data structures," *ACM Computing Surveys*, 16, June 1984

[Same85] Samet, H. and R. E. Webber, "Storing A collection of polygons using quadtrees," *ACM Transactions on Graphics 4, 3*(July 1985), pp. 182-222 (also *Proceedings of Computer Vision and Pattern Recognition 83*, Washington, DC, June 1983, pp. 127-132; and University of Maryland Computer Science TR-1372).

[Same88] Samet, H., "An overview of quadtrees, octrees, and related hierachical data structures," In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD.* NATO ASI Series F, vol.40, pp. 51-68. Springer-Verlag, 1988

[Same90a] Samet, H., "Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS," Addison-Wesley, Reading, MA, 1990

[Same90b] Samet, H., "The Design and Analysis of Spatial Data Structures," Addison Wesley, Reading, Mass., 1990

[Seeg90] Seeger, B. and H.-P. Kriegel, "The buddy-tree: an efficient and robust access method for spatial data base systems," In *Proc. 16th Int. Conference on Very Large Data Bases*, pp. 590-601, 1990

[Seeg91] Seeger, B., "Performance comparison of segment access methods implemented on top of buddy tree." In O. Gunther and H. Schek, Advanced in Spatial Databases, Number 525 in LNCS, Berlin/Heidelberg/New York, pp. 277-296, 1991

[Sell87] Sellis, T., N. Roussopoulos, and C. Faloutsos, "The $R^+$-tree: A dynamic index for multidimensional objects," In *Proc. 13th Int. Conference on Very Large Data Bases*, pp. 507-518, 1987

[Six88] Six, H. and P. Widmayer, "Spatial searching in geometric databases," In *Proc. 4th IEEE Int. Conf. On Data Eng.*, pp. 496-503.

[Smit90] Smith, T.R., and P. Gao, "Experimental performance evaluations on spatial access methods." In Proc. 4th Int. Symp. on Spatial Data Handling, Zurich, pp.991-1002, 1990

[Suth74] Sutherland, I. E., R. F. Sproull, and R. A. Schumacker, "A characterization of ten hidden-surface algorithms," *ACM Computing Surveys* 6, 1(March 1974), pp. 1-55, 1974

[Tamm82] Tamminen, M., "The extendible cell method for closest point problems," *BIT* 22, pp. 27-41, 1982

[Tamm83] Tamminen, M., "Performance anlysis of cell based geometric file organizations," International Journal of Computer Vision, Graphics and Image Processing 24, pp. 160-181, 1983

[Whan85] Whang, K. Y. and R. Krishnamurthy, "Multilevel grid files," Yorktown Heights, NY: IBM Research Laboratory, 1985