Multidimensional Access Methods*

Volker Gaede and Oliver Günther
Institut für Wirtschaftsinformatik
Humboldt-Universität zu Berlin
Spandauer Str. 1
10178 Berlin, Germany
{gaede,guenther}@wiwi.hu-berlin.de

Abstract

Search operations in databases require some special support at the physical level. This is true for conventional databases as well as for spatial databases, where typical search operations include the *point query* (find all objects that contain a given search point) and the *region query* (find all objects that overlap a given search region). More than ten years of spatial database research have resulted in a great variety of multidimensional access methods to support such operations. This paper gives an overview of that work. After a brief survey of spatial data management in general, we first present the class of *point access methods*, which are used to search sets of points in two or more dimensions. The second part of the paper is devoted to *spatial access methods* to handle extended objects, such as rectangles or polyhedra. We conclude with a discussion of theoretical and experimental results concerning the relative performance of various approaches.

Keywords: multidimensional access methods, data structures, spatial databases

*This work was partially supported by the German Research Society (DFG/SFB 373) and by the ESPRIT Working Group CONTESSA (8666).

Contents

1 Introduction

With an increasing number of computer applications that rely heavily on multidimensional data, the database community has recently devoted considerable attention to spatial data management. While the main motivation originated from the geosciences and mechanical CAD, the range of possible applications has expanded to areas such as robotics, visual perception and autonomous navigation, environmental protection, and medical imaging (Günther and Buchmann 1990).

Just as broad as the range of applications is the range of interpretations given to the term *spatial data management*. In VLSI CAD and cartography, this term refers to applications that rely mostly on two–dimensional or layered two–dimensional data. VLSI data is usually represented by rectilinear polylines or polygons whose edges are iso-oriented, i.e., parallel to the coordinate axes. Typical operations include intersection and geometric routing (Shekhar and Liu 1995). Cartographic data is also two–dimensional with points, lines, and regions as basic primitives. In contrast to VLSI CAD, however, the shapes are often characterized by extreme irregularities. Common operations include spatial searches, map overlay, as well as distance-related operations. In mechanical CAD, on the other hand, data objects are usually three–dimensional solids. They may be represented in a variety of data formats, including cell decomposition schemes, constructive solid geometry (CSG), and boundary representations (Kemper and Wallrath 1987). Yet other applications emphasize the processing of unanalyzed images, such as X–rays and satellite imagery, from which features are extracted. In those areas, the terms *spatial database* and *image database* are sometimes even used interchangeably.

Strictly speaking, however, *spatial databases* contain multidimensional data with explicit knowledge about objects, their extent, and their position in space. The objects are usually represented in some vector–based format, and their relative position may be explicit or implicit (i.e., derivable from the internal representation of their absolute positions). *Image databases*, on the other hand, often place less emphasis on data analysis. They provide storage and retrieval for unanalyzed pictorial data, which is typically represented in some raster format. Techniques developed for the storage and manipulation of image data can be applied to other media as well, such as infrared sensor signals or sound.

In this survey, we assume that the goal is to manipulate analyzed multidimensional data, and that unanalyzed images are only handled as the source from which spatial data can be derived. The challenge for the developers of a spatial database system lies not so much in providing yet another collection of special–purpose data structures. One rather has to find abstractions and architectures to implement generic systems, i.e., to build systems with generic spatial data management capabilities that can be tailored to the requirements of a particular application domain. Important issues in this context include the handling of spatial representations and data models, multidimensional access methods, as well as pictorial or spatial query languages and their optimization.

This paper is a survey of multidimensional access methods to support search operations in spatial databases. Figure 1 gives a first overview of the diversity of existing multidimensional access methods. The goal of this paper is not to describe all of these

structures, but to discuss the most prominent ones, to present possible taxonomies, and to establish references to further literature.
Several shorter surveys have been published previously in various Ph.D. theses (Ooi 1990; Kolovson 1990; Oosterom 1990; Schiwietz 1993). Widmayer (1991) gives an overview of work published before 1991. Like the thesis by Schiwietz, however, his survey is only available in German. Samet's books (1989, 1990) present the state of the art until 1989. However, they primarily cover quadtrees and related data structures. Lomet (1991) discusses the field from a systems-oriented point of view.

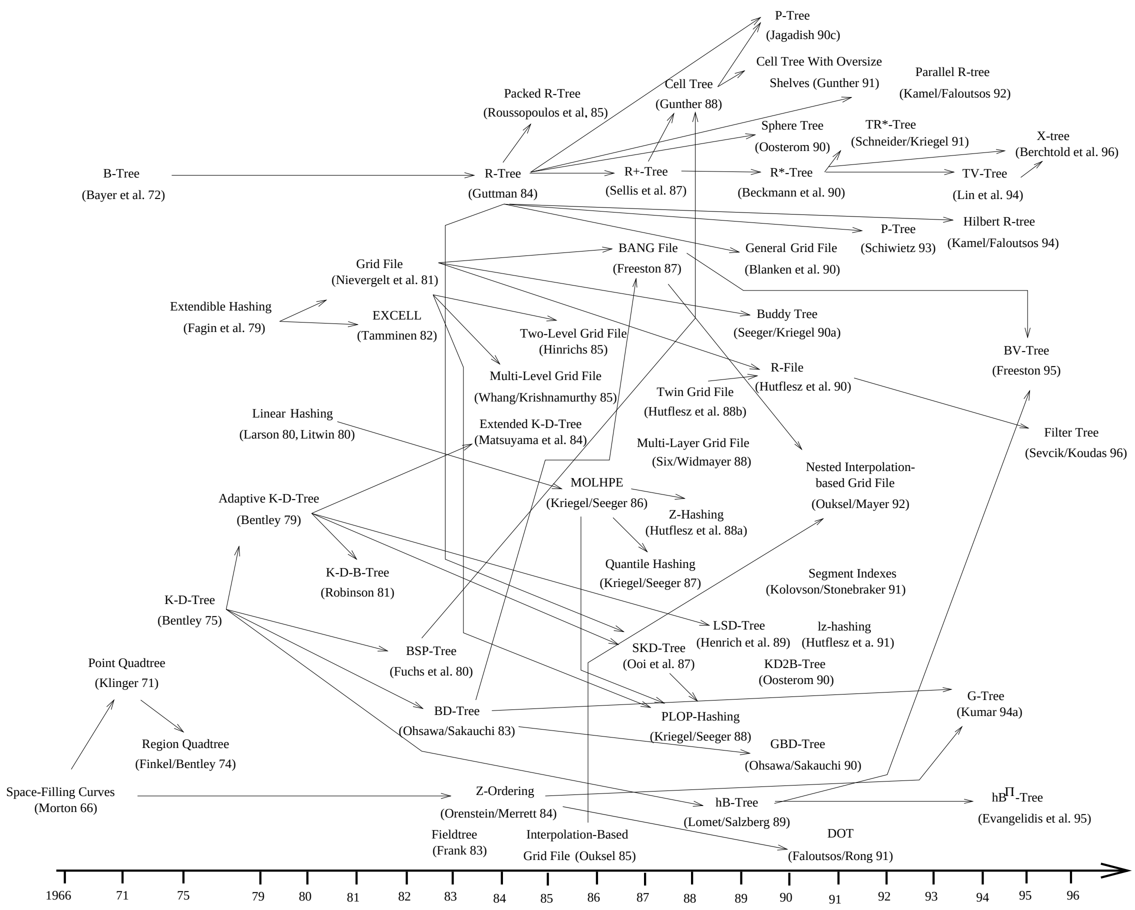


Figure 1: History of Multi-Dimensional Access Methods Following (Lu and Ooi 1993).

The remainder of the paper is organized as follows. Section 2 discusses some basic properties of spatial data and their implications on the design and implementation of spatial databases. Section 3 gives an overview of some traditional data structures that had an impact on the design of multidimensional access methods. Sections 4 and 5 form the core of this survey, presenting a variety of point access methods (PAMs) and spatial access methods (SAMs), respectively. Some remarks about theoretical and experimental analyses are contained in Section 6, and Section 7 concludes the paper.

2 Organization of Spatial Data

2.1 What Is Special About Spatial?

To obtain a better understanding of the requirements in spatial database systems, we first discuss some basic properties of spatial data. First, spatial data has a *complex structure*. A spatial data object may be composed of a single point or several thousands of polygons, arbitrarily distributed across space. It is usually not possible to store collections of such objects in a single relational table with a fixed tuple size. Second, spatial data is often *dynamic*. Insertions and deletions are interleaved with updates, and data structures used in this context have to support this dynamic behavior without deteriorating over time. Third, spatial databases tend to be *large*. Geographic maps, for example, typically occupy several gigabytes of storage. The integration of secondary and tertiary memory is therefore essential for efficient processing (Chen et al. 1995). Fourth, there is *no standard algebra* defined on spatial data, although several proposals have been made in the past (Egenhofer 1989; Güting 1989; Scholl and Voisard 1989; Güting and Schneider 1993). This means in particular that there is no standardized set of base operators. The set of operators heavily depends on the given application domain, although some operators (such as intersection) are more common than others. Fifth, many spatial operators are *not closed*. The intersection of two polygons, for example, may return any number of single points, dangling edges, or disjoint polygons. This is particularly relevant when operators are applied consecutively. Sixth, although the computational costs vary among spatial database operators, they are generally *more expensive* than standard relational operators.

An important class of geometric operators that needs special support at the physical level is the class of *spatial search operators*. Retrieval and update of spatial data is usually based not only on the value of certain alphanumeric attributes, but also on the spatial location of a data object. A retrieval query on a spatial database often requires the fast execution of a geometric search operation such as a *point* or *region query*. Both operations require fast access to those data objects in the database that occupy a given location in space.

To support such search operations, one needs special *multidimensional access methods*. The main problem for the design of such methods, however, is that *there exists no total ordering among spatial objects that preserves spatial proximity.* In other words, there is no mapping from two- or higher-dimensional space into one-dimensional space such that any two objects that are spatially close in the higher-dimensional space are also close to each other in the one-dimensional sorted sequence.

This makes the design of efficient access methods in the spatial domain much more difficult than in traditional databases, where a broad range of efficient and well-understood access methods is available. Examples for such *one-dimensional access methods* (also called *single key structures*, although that term is somewhat misleading) include the B-Tree (Bayer and McCreight 1972) and extendible hashing (Fagin et al. 1979); see Section 3.1 for a brief discussion. A popular approach in many commercial database systems to handle multidimensional search queries consists in the consecutive application of such single key structures, one per dimension. Unfortunately, this approach can be very inefficient (Kriegel 1984). Since each index is traversed indepen-

dently of the others, we cannot exploit the possibly high selectivity in one dimension for narrowing down the search in the remaining dimensions. In general, there is no easy and obvious way to extend single key structures in order to handle multidimensional data.
There is a variety of requirements that multidimensional access methods should meet, based on the properties of spatial data and their applications (Robinson 1981; Lomet and Salzberg 1989; Nievergelt 1989):

1. *Dynamics.* As data objects are inserted and deleted from the database in any given order, access methods should continuously keep track of the changes.

2. *Secondary/tertiary storage management.* Despite growing main memories, it is often not possible to hold the complete database in main memory. Therefore, access methods need to integrate secondary and tertiary storage in a seamless manner.

3. *Broad range of supported operations.* Access methods should not support just one particular type of operation (such as retrieval) at the expense of other tasks (such as deletion).

4. *Independence of the input data and insertion sequence.* Access methods should maintain their efficiency even when input data is highly skewed or the insertion sequence is changed. This point is especially important for data that is distributed differently along the various dimensions.

5. *Simplicity.* Intricate access methods with many special cases are often error-prone to implement and thus not sufficiently robust to be used in large-scale applications.

6. *Scalability.* Access methods should adapt well to database growth.

7. *Time efficiency.* Spatial searches should be fast. A major design goal is to meet the performance characteristics of one-dimensional B-trees: First, access methods should guarantee a logarithmic worst-case search performance for *all* possible input data distributions regardless of the insertion sequence. Second, this worst-case performance should hold for any combination of the d attributes.

8. *Space efficiency.* An index should be small in size compared with the data to be addressed and therefore guarantee a certain storage utilization.

9. *Concurrency and recovery.* In modern databases where multiple users concurrently update, retrieve and insert data, access methods should provide robust techniques for transaction management without significant performance penalties.

10. *Minimum impact.* The integration of an access method into a database system should have minimum impact on existing parts of the system.

2.2 Definitions and Queries

We have already introduced the term *multidimensional access methods* to denote the large class of access methods that support searches in spatial databases and that are the subject of this survey. Within this class, we distinguish between *point access methods (PAMs)* and *spatial access methods (SAMs)*. Point access methods have primarily been designed to perform spatial searches on point databases (that is, databases that store only points). The points may be embedded in two or more dimensions, but they do not have a spatial extension. Spatial access methods, on the other hand, are able to manage extended objects, such as lines, polygons, or even higher-dimensional polyhedra. In the literature, one often finds the term *spatial access method* referring to what we call *multidimensional access method*. Other terms used for this purpose include *spatial index* or *spatial index structure*.

We generally assume that the given objects are embedded in d-dimensional Euclidean space E^d or a suitable subspace thereof. In this paper, this space is also referred to as *universe* or *original space*. Any point object stored in a spatial database has a unique location in the universe, defined by its d coordinates. Unless the distinction is essential, we use the term *point* both for locations in space and for point objects stored in the database. Note, however, that any point in space can be occupied by several point objects stored in the database.

A *(convex) d–dimensional polytope* P in E^d is defined to be the intersection of some finite number of closed halfspaces in E^d, such that the dimension of the smallest affine subspace containing P is d. If $a \in E^d - \{0\}$ and $c \in E^1$ then the $(d-1)$–dimensional set $H(a,c) = \{x \in E^d : x \cdot a = c\}$ defines a *hyperplane* in E^d. A hyperplane $H(a,c)$ defines two closed halfspaces, the *positive halfspace* $1 \cdot H(a,c) = \{x \in E^d : x \cdot a \geq c\}$, and the *negative halfspace* $-1 \cdot H(a,c) = \{x \in E^d : x \cdot a \leq c\}$. A hyperplane $H(a,c)$ *supports* a polytope P if $H(a,c) \cap P \neq \emptyset$ and $P \subseteq 1 \cdot H(a,c)$, i.e., if $H(a,c)$ embeds parts of P's boundary. If $H(a,c)$ is any hyperplane supporting P then $P \cap H(a,c)$ is a *face* of P. The faces of dimension 1 are called *edges*; those of dimension 0 *vertices*.

By forming the union of some finite number of polytopes $Q_1, ..., Q_n$, we obtain a *(d–dimensional) polyhedron* Q in E^d that is not necessarily convex. Following the intuitive understanding of polyhedra, we require that the Q_i $(i = 1, ..., n)$ have to be connected. Note that this still allows for *polyhedra with holes*. Each face of Q is either the face of some Q_i, or a fraction thereof, or the result of the intersection of two or more Q_i. Each polyhedron P divides the points in space into three subsets that are mutually disjoint: its *interior*, its *boundary*, and its *exterior*.

Following usual conventions, we use the terms *line* and *polyline* to denote a one-dimensional polyhedron and the terms *polygon* and *region* to denote a two-dimensional polyhedron. We further assume that for each k $(0 \leq k \leq d)$, the set of k-dimensional polyhedra forms a data type, which leads us to the common collection of *spatial data types* $\{$*Point, Line, Region, ...*$\}$. Combined types sometimes also occur. An object o in a spatial database is usually defined by several non-spatial attributes and one attribute of some spatial data type. This spatial attribute describes the object's *spatial extent* $o.G$. In the spatial database literature, the terms *geometry*, *shape*, and *spatial extension* are often used instead of *spatial extent*. For the *description* of $o.G$ one finds the terms *shape* *descriptor, shape description*, *shape information*, and *geometric description*, among others.

Indices often perform more efficiently when handling simple entries of about equal size. One therefore often abstracts from the actual shape of a spatial object before inserting it into an index. This can be achieved by *approximating* the original data object with a simpler shape, such as a bounding box or a sphere. Given the minimum bounding interval $I_i(o) = [l_i, u_i]$ ($l_i, u_i \in E^1$) describing the extent of the spatial object o along dimension i, the d-dimensional *minimum bounding box (MBB)* is defined by $I^d(o) = I_1(o) \times I_2(o) \times ... \times I_d(o)$.

An index may only administer the MBB of each object, together with a pointer to the object's database entry (the *object ID* or *object reference*). With this design, the index only produces a set of *candidate solutions* (Figure 2). For each candidate obtained during this *filter step*, we have to decide whether the MBB is sufficient to guarantee that the object itself satisfies the search predicate. In those cases, the object can be added directly to the query result (dashed line). However, there are often cases where the MBB does not prove to be sufficient. In a *refinement step* we then have to retrieve the exact shape information from secondary memory and test it against the predicate. If the predicate evaluates to true, the object is added to the query result, otherwise we have a *false drop*.

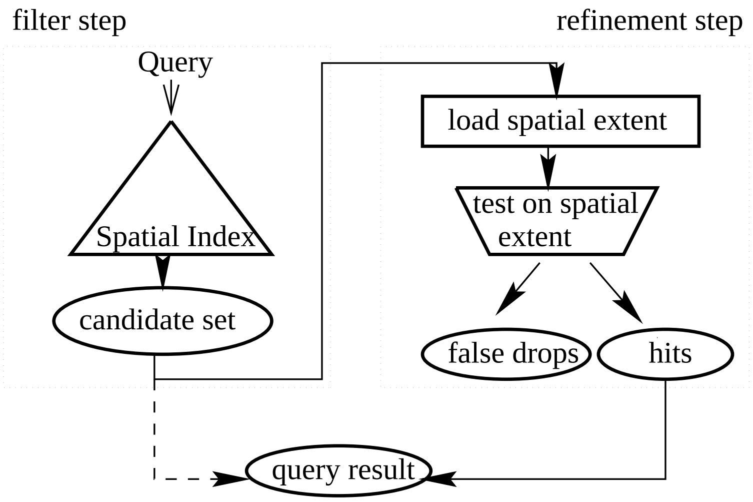


Figure 2: Multi-Step Spatial Query Processing Following (Brinkhoff et al. 1994)

Another way of obtaining simple index entries is to represent the shape of each data object as the geometric union of simpler shapes (such a convex polygons with a bounded number of vertices). This approach is called *decomposition.*

We have mentioned the term *efficiency* several times so far without giving a formal definition. In the case of space efficiency, this can easily be done: The goal is to minimize the number of bytes occupied by the index. For time efficiency the situation is not so clear. Elapsed time is obviously what the user cares about, but one should keep in mind that the corresponding measurements depend a lot on implementation, hardware utilization, and other details. In the literature, one therefore often finds a seemingly more objective performance measure: the number of disk accesses performed during a search. This approach, which has become popular with the B-tree, is based on the assumption that most searches are I/O-bound rather than CPU-bound – an assumption that is not always true in spatial data management, however. In applications where objects have complex shapes, the refinement step can incur major CPU costs and change the balance with I/O (Gaede 1995b; Hoel and Samet 1995). Of course, one should keep the minimization of the number of disk accesses in mind as *one* design goal. Practical evaluations, however, should always give some information on elapsed times and the conditions under which they were achieved.

As noted above, in contrast to relational databases there exists neither a standard spatial algebra nor a standard spatial query language. The set of operators strongly depends on the given application domain, although some operators (such as intersection) are generally more common than others. Queries are often expressed by some extension of SQL that allows abstract data types to represent spatial objects and their associated operators (Roussopoulos and Leifker 1984; Egenhofer 1994). The result of a query is usually a set of spatial data objects. In the remainder of this section, we give a formal definition of several of the more common spatial database operators. Figures 3 through 8 give some concrete examples.

Query 1 (Exact Match Query EMQ, Object Query) *Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o with the same spatial extent as o'.*

$$EMQ(o') = \{o | o'.G = o.G\}$$

Query 2 (Point Query PQ) *Given a point $p \in E^d$, find all objects o overlapping p.*

$$PQ(p) = \{o | p \cap o.G = p\}$$

The point query can be regarded as a special case of several of the following queries, such as the intersection query, the window query, or the enclosure query.

Query 3 (Window Query WQ, Range Query) *Given a d-dimensional interval $I^d = [l_1, u_1] \times [l_2, u_2] \times ... \times [l_d, u_d]$, find all objects o having at least one point in common with I^d.*

$$WQ(I^d) = \{o | I^d \cap o.G \neq \emptyset\}$$

The query implies that the window is iso-oriented, i.e., its faces are parallel to the coordinate axes. A more general variant is the *region query* that permits search regions to have arbitrary orientations and shapes:

Query 4 (Intersection Query IQ, Region Query, Overlap Query) *Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o having at least one point in common with o'.*

$$IQ(o') = \{o | o'.G \cap o.G \neq \emptyset\}$$

Query 5 (Enclosure Query EQ) *Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o enclosing o'.*

$$EQ(o') = \{o|(o'.G \cap o.G) = o'.G\}$$

Query 6 (Containment Query CQ) *Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o enclosed by o'.*

$$CQ(o') = \{o|(o'.G \cap o.G) = o.G\}$$

The enclosure and the containment query are symmetric to each other. They are both more restrictive formulations of the intersection query by specifying the result of the intersection to be one of the two inputs.

Query 7 (Adjacency Query AQ) *Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o adjacent to o'.*

$$AQ(o') = \{o|o.G \cap o'.G \neq \emptyset \wedge o'.G^\circ \cap o.G^\circ = \emptyset\}$$

Here, $o'.G^\circ$ and $o.G^\circ$ denote the interiors of the spatial extents $o'.G$ and $o.G$, respectively.

Query 8 (Nearest Neighbor Query NNQ) *Given an object o' with spatial extent $o'.G \subseteq E^d$, find all objects o having a minimum distance from o'.*

$$NNQ(o') = \{o|\forall o'' : \ dist(o'.G, o.G) \leq dist(o'.G, o''.G)\}$$

Distance between extended spatial data objects is usually defined as the distance between their closest points. Common distance functions for points include the Euclidean and the Manhattan distance.

Figure 3: Point Query

Figure 4: Window Query

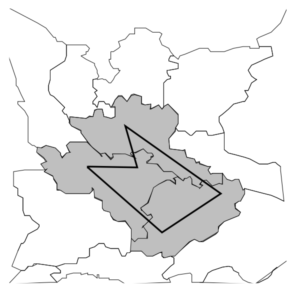


Figure 5: Intersection Query

Besides spatial selections, as exemplified by Queries 1 through 8, the *spatial join* is one of the most important spatial operations and can be defined as follows (Günther 1993):

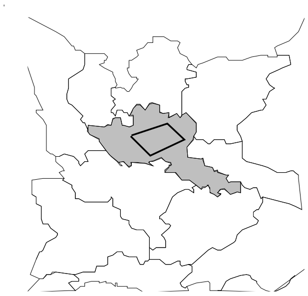


Figure 6: Enclosure Query

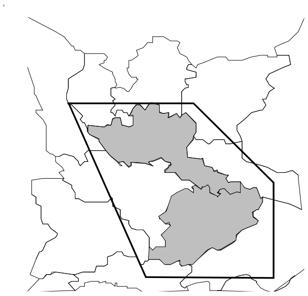


Figure 7: Containment Query

Figure 8: Adjacency Query

Query 9 (Spatial Join) *Given two collections R and S of spatial objects and a spatial predicate θ, find all pairs of objects $(o, o') \in R \times S$ where $\theta(o.G, o'.G)$ evaluates to true.*

$$R \bowtie_\theta S = \{(o, o') | o \in R \wedge o' \in S \wedge \theta(o.G, o'.G)\}$$

As for the spatial predicate θ, a brief survey of the literature (Orenstein 1986; Becker 1992; Rotem 1991; Günther 1993; Brinkhoff, Kriegel, and Seeger 1993; Gaede and Riekert 1994; Brinkhoff 1994; Lo and Ravishankar 1994; Aref and Samet 1994; Papadias, Theodoridis, Sellis, and Egenhofer 1995) yields a wide variety of possibilities, including

- $intersects(\cdot)$
- $contains(\cdot)$
- $is\_enclosed\_by(\cdot)$
- $distance(\cdot)\Theta q$, with $\Theta \in \{=, \leq, <, \geq, >\}$ and $q \in E^1$
- $northwest(\cdot)$
- $adjacent(\cdot)$
- $meets(\cdot)$

A closer inspection of these spatial predicates shows that the *intersection join* $R \bowtie_{intersects} S$ plays a crucial role for the computation in virtually all of those cases (Gaede and Riekert 1994). For predicates such as *contains, encloses,* or *adjacent*, for example, the intersection join is an efficient filter that yields a set of candidate solutions typically much smaller than the Cartesian product $R \times S$.

3 Basic Data Structures

3.1 One-Dimensional Access Methods

Classical one-dimensional access methods are an important foundation for almost all multidimensional access methods. Although the survey on hashing functions by Knott

(1975) is somewhat dated, it represents a good coverage of the different approaches. In practice, the most common one-dimensional structures include linear hashing (Litwin 1980; Larson 1980), extendible hashing (Fagin et al. 1979), and the B-tree (Bayer and McCreight 1972; Comer 1979). Hierarchical access methods such as the B-tree are scalable and behave well in the case of skewed input; they are nearly independent of the distribution of the input data. This is not necessarily true for hashing techniques, whose performance may degenerate depending on the given input data and hash function. This problem is aggravated by the use of *order-preserving* hash functions (Orenstein 1983; Garg and Gotlieb 1986) that try to preserve neighborhood relationships between data items, in order to support range queries. As a result, highly skewed data keeps accumulating at a few selected locations in image space.

3.1.1 Linear Hashing (Larson 1980; Litwin 1980)

Linear hashing divides the universe $[A, B)$ of possible hash values into binary intervals of size $(B-A)/2^k$ or $(B-A)/2^{k+1}$ for some $k \geq 0$. Each interval corresponds to a *bucket*, i.e., a collection of records stored on a disk page. $t \in [A, B)$ is a pointer that separates the smaller intervals from the larger ones: all intervals of size $(B-A)/2^k$ are to the left of t, and vice versa. If a bucket reaches its capacity due to an insertion, the interval $[t, t+(B-A)/2^k)$ is split into two subintervals of equal size, and t is advanced to the next large interval remaining. Note that the split interval does not have to be the same interval as the one that caused the split; consequently, there is no guarantee that the split relieves the bucket in question from its overload. If an interval contains more objects than bucket capacity permits, the overload is stored on an overflow page, which is linked to the original page. When $t = B$, the file has doubled and all intervals have the same length $(B-A)/2^{k+1}$. In this case we reset the pointer t to A and resume the split procedure for the smaller intervals.

3.1.2 Extendible Hashing (Fagin et al. 1979)

As linear hashing, *extendible hashing* organizes the data in binary intervals, here called *cells*. Overflow pages are avoided in extendible hashing by using a central *directory*. Each cell has an index entry in that directory; it initially corresponds to one bucket. If during an insertion a bucket at maximal depth exceeds its maximum capacity, all cells are split into two. New index entries are created and the directory doubles in size. Since not each bucket was at full capacity before the split, it may now be possible to fit more than one cell in the same bucket. In that case, adjacent cells are regrouped in *data regions* and stored on the same disk page. In the case of skewed data this may lead to a situation where numerous directory entries exist for the same data region (and therefore the same disk page). Even in the case of uniformly distributed data, the directory growth may be superlinear (Flajolet 1983). Exact match searches take no more than two page accesses: one for the directory and one for the bucket with the data. This is more than the best-case performance of linear hashing, but better than the worst case.

Besides the potentially poor space utilization of the index, extendible hashing also suffers from a non-incremental growth of the index due to the doubling steps. To address these problems, Lomet (1983) proposed a technique called *bounded index extendible hashing*. In this proposal, the index grows as in extendible hashing until its size reaches a *predetermined maximum*, i.e., the index size is bounded. Once this limit is reached while inserting new items, bounded index extendible hashing starts *doubling the data bucket size* rather than the index size.

3.1.3 The B-Tree (Bayer and McCreight 1972)

Other than hashing schemes, the *B-tree* and its variants (Comer 1979) organize the data in a hierarchical manner. B-trees are balanced trees that correspond to a nesting of intervals. Each node ν corresponds to a disk page $D(\nu)$ and an interval $I(\nu)$. If ν is an interior node then the intervals $I(\nu_i)$ corresponding to the immediate descendants of ν are mutually disjoint subsets of $I(\nu)$. Leaf nodes contain pointers to data items; depending on the type of B-tree, interior nodes may do so as well. B–trees have an upper and lower bound for the number of descendants of a node. The lower bound prevents the degeneration of trees and leads to an efficient storage utilization. Nodes whose number of descendants drops below the lower bound are deleted and its content is distributed among the adjacent nodes at the same tree level. The upper bound follows from the fact that each tree node corresponds to exactly one disk page. If during an insertion a node reaches its capacity, it is split into two. Splits may propagate up the tree. As the size of the intervals depends on the given data (and the insertion sequence), the B-tree is an adaptive data structure. For uniformly distributed data, however, extendible as well as linear hashing outperform the B-tree on the average for exact match queries, insertions and deletions.

3.2 Main Memory Structures

Early multidimensional access methods did not account for paged secondary memory and are therefore less suited for large spatial databases. In this section, we review several of these fundamental data structures, which are adapted and incorporated in numerous multidimensional access methods. To illustrate the methods, we introduce a small scenario that we shall use as a running example throughout this survey. The scenario, depicted in Figure 9, contains ten points *pi* and ten polygons *ri*, randomly distributed in a finite two-dimensional universe. To represent polygons, we shall often use their centroids *ci* (not pictured) or their minimum bounding boxes (MBBs) *mi*. Note that the quality of the MBB approximation varies considerably. The MBB m8, for example, provides a fairly tight fit, whereas r5 is only about half as large as its MBB m5.

3.2.1 The K-D-Tree (Bentley 1975; Bentley 1979)

One of the most prominent d-dimensional data structures is the *k-d-tree*. The k-d-tree is a binary search tree that represents a recursive subdivision of the universe into subspaces by means of $(d-1)$-dimensional hyperplanes. The hyperplanes are iso-oriented, and their direction alternates between the d possibilities. For $d = 3$, for example, splitting hyperplanes are alternately perpendicular to the x-, y-, and z-axis.

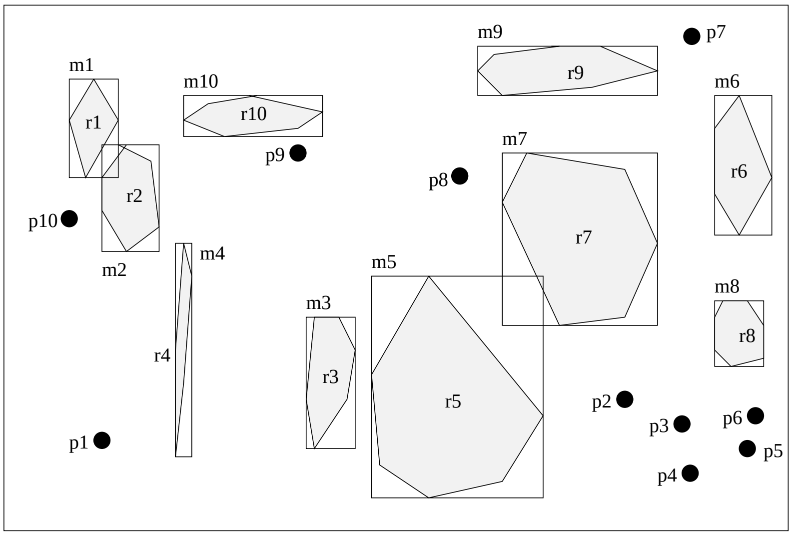


Figure 9: Running Example

Each splitting hyperplane has to contain at least one data point, which is used for its representation in the tree. Interior nodes have one or two descendants each and function as a discriminator to guide the search. Searching and insertion of new points are straightforward operations. Deletion is somewhat more complicated and may cause a reorganization of the subtree below the data point to be deleted.

Figure 10 shows a k-d-tree for the running example. Because the tree can only handle points, we represent the polygons by their centroids *ci*. The first splitting line is the vertical line crossing c3. We therefore store c3 in the root of the corresponding k-d-tree. The next splits occur along horizontal lines crossing p10 (for the left subtree) and c7 (for the right subtree), and so on.

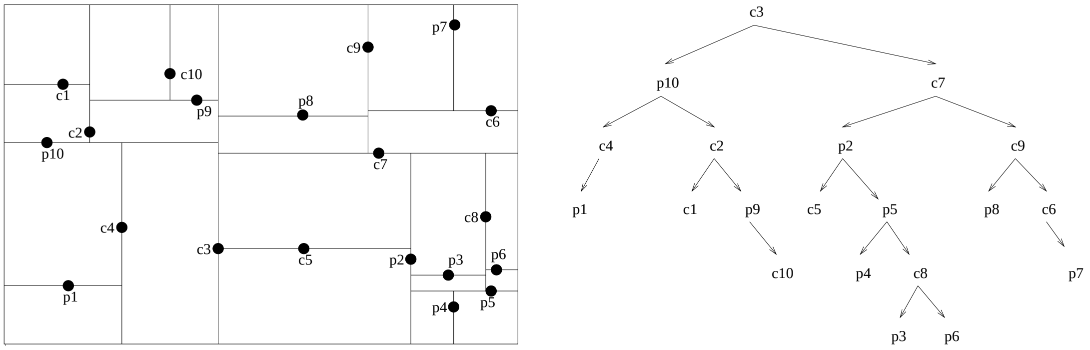


Figure 10: K-D-Tree

One disadvantage of the k-d-tree is that the structure is sensitive to the order in which the points are inserted. Another one is that data points are scattered all over the tree. The *adaptive k-d-tree* (Bentley and Friedman 1979) mitigates these problems by choosing a split such that one finds about the same number of elements on both sides. While the splitting hyperplanes are still parallel to the axes, they do not have to contain a data point and their directions do not have to be strictly alternating anymore. As a result, the split points are not part of the input data; all data points are stored in the leaves. Interior nodes contain the dimension (e.g. x or y) and the coordinate of the corresponding split. Splitting is continued recursively until each subspace contains only a certain number of points. The adaptive k-d-tree is a rather static structure; it is obviously difficult to keep the tree balanced in the presence of frequent insertions and deletions. The structure works best if all the data is known a priori and if updates are rare. Figure 11 shows an adaptive k-d-tree for the running example. Note that the tree still depends on the order of insertion.

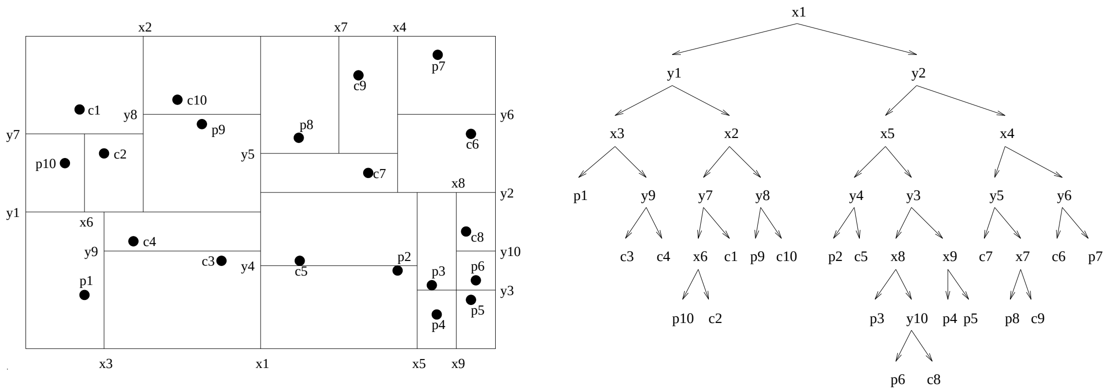


Figure 11: Adaptive K-D-Tree

Another variant of the k-d-tree is the *bintree* (Tamminen 1984). This structure partitions the universe recursively into d-dimensional boxes of *equal size* until each one contains only a certain number of points. Even though this kind of partitioning is less adaptive, it has several advantages, such as the implicit knowledge of the partitioning hyperplanes. In the remainder of this paper, we shall encounter several other structures that are based on this kind of partitioning.

A disadvantage common to all k-d-trees is that for certain distributions no hyperplane can be found which splits the data points evenly (Lomet and Salzberg 1989). By introducing a more flexible partitioning scheme, the BSP-tree presented subsequently avoids this problem completely.

3.2.2 The BSP-Tree (Fuchs, Kedem, and Naylor 1980; Fuchs, Abram, and Grant 1983)

Splitting the universe only along iso-oriented hyperplanes is a severe restriction in the schemes presented so far. Allowing arbitrary orientations gives more flexibility to find a hyperplane that is well-suited for the split. A well–known example for such a method is the *binary space partitioning (BSP) tree*. Like k-d-trees, BSP-trees are binary trees that represent a recursive subdivision of the universe into subspaces by means of $(d-1)$–dimensional hyperplanes. Each subspace is subdivided independently of its history and of the other subspaces. The choice of the partitioning hyperplanes depends on the distribution of the data objects in a given subspace. The decomposition usually continues until the number of objects in each subspace is below a given threshold.

The resulting partition of the universe can be represented by a BSP-tree, where each hyperplane corresponds to an interior node of the tree and each subspace corresponds to a leaf. Each leaf stores references to those objects that are contained in the corresponding subspace. Figure 12 shows a BSP-tree for the running example with no more than two objects per subspace.

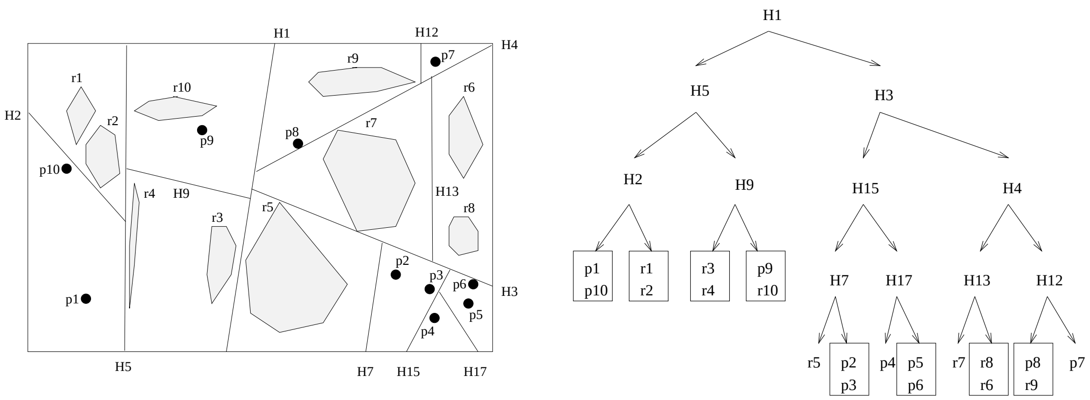


Figure 12: BSP-Tree

In order to perform a point query, we insert the search point into the root of the tree and determine on which side of the corresponding hyperplane it is located. Next, we insert the point into the corresponding subtree and proceed recursively until we reach a leaf of the tree. Finally, we have to examine the data objects in the corresponding subspace whether they contain the search point. The range search algorithm is a straightforward generalization.

BSP-trees can adapt well to different data distributions. However, they are typically not balanced and may have very deep subtrees, which has a negative impact on tree performance. BSP-trees also have higher space requirements, since storing an arbitrary hyperplane per split occupies more storage space than a discriminator, which is typically just a real number.

3.2.3 The BD-Tree (Ohsawa and Sakauchi 1983)

The *BD-tree* (Ohsawa and Sakauchi 1983) is a binary tree representing a subdivision of the data space into interval-shaped regions. Each of those regions is encoded in a bit string and associated with one of the BD-tree nodes. Here, these bit strings are called *DZ-expressions*; they are also known as *Peano codes*, *ST_MortonNumber*, or *z-values* (cf. Section 5.1.2).

Given a region R, one computes the corresponding DZ-expression as follows. For simplicity we restrict this presentation to the two-dimensional case; we also assume that the first subdividing hyperplane is a vertical line. If R lies to the left of that line, the first bit of the corresponding DZ-expression is `0`, otherwise it is `1`. In the next step we subdivide the subspace containing R by a horizontal line. If R lies below that line, the second bit of the DZ-expression is `0`, otherwise it is `1`. As this decomposition progresses, we obtain one bit per splitting line. Bits at odd positions refer to vertical lines, and bits at even positions to horizontal lines, which explains why this scheme is often referred to as *bit interleaving.*

To avoid the storage utilization problems that are often associated with a strictly regular partitioning, the BD-tree employs a more flexible splitting policy. Here one can split a node by cutting out an interval-shaped excision from the corresponding region. The two child nodes of the node to be split will then have different interpretations: one represents the excision, the other one represents the remainder of the original region. Note that the remaining region is no longer interval-shaped. With this policy, the BD-tree can guarantee that, after node splitting, each of the data buckets contains at least one third of the original entries.

Figure 13 shows a BD-tree for the running example. In the case of an excision, it is always represented by the left child of the node that was split.

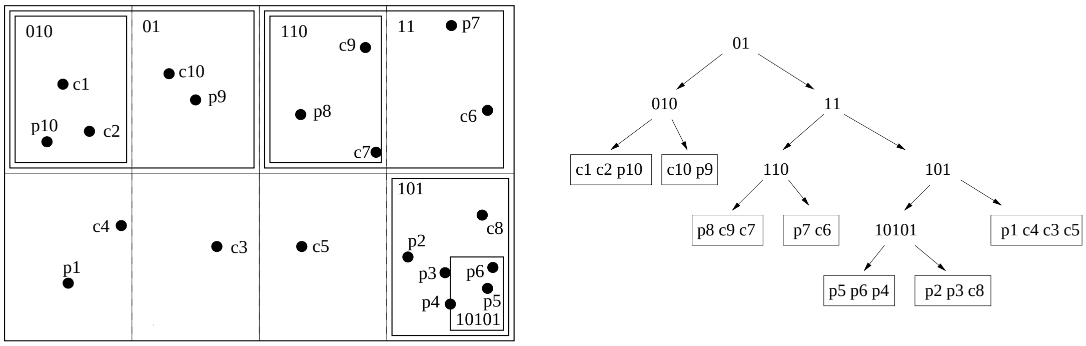


Figure 13: BD-Tree

For exact match we first compute the full bit-interleaved prefix of the search record. Starting from the root, we recursively compare this prefix with the stored DZ-expressions of each internal node. If it matches, we follow the corresponding link, otherwise we follow the other link until we reach the leaf level of the BD-tree.

Ohsawa and Sakauchi do not describe how the directory should be maintained on disk. This was done later by Dandamudi and Sorenson (1986, 1991), who also presented several other proposals for improving BD-tree performance.

3.2.4 The Quadtree

The quadtree with its many variants is a close relative of the k-d-tree. For an extensive discussion of this structure, see (Samet 1984; Samet 1989; Samet 1990). While the term *quadtree* usually refers to the two-dimensional variant, the basic idea applies to arbitrary d. Like the k-d-tree, the quadtree decomposes the universe by means of iso-oriented hyperplanes. An important difference, however, is the fact that quadtrees are not binary trees anymore. In d dimensions, the interior nodes of a quadtree have 2^d descendants, each corresponding to an interval-shaped partition of the given subspace. These partitions do not have to be of equal size, although that is often the case. For $d = 2$, for example, each interior node has four descendants, each corresponding to a rectangle. These rectangles are typically referred to as the NW, NE, SW, and SE quadrants. The decomposition into subspaces is usually continued until the number of objects in each partition is below a given threshold. Quadtrees are therefore not necessarily balanced; subtrees corresponding to densely populated regions may be deeper than others.

Searching in a quadtree is similar to searching in an ordinary binary search tree. At each level, one has to decide which of the four subtrees need to be included in the future search. In the case of a point query, typically only one subtree qualifies, whereas for range queries there are often several. We repeat this search step recursively until we reach the leaves of the tree.

Finkel and Bentley (1974) proposed one of the first variants: It is called *point quadtree* and is essentially a multidimensional binary search tree. The point quadtree is constructed consecutively by inserting the data points one by one. For each point, we first perform a point search. If we do not find the point in the tree, we insert it into the leaf node where the search has terminated. The corresponding partition is divided into 2^d subspaces with the new point at the center. The deletion of a point requires the restructuring of the subtree below the corresponding quadtree node. A simple way to achieve this is to reinsert all points into the subtree. Figure 14 shows a two-dimensional point quadtree for the running example.

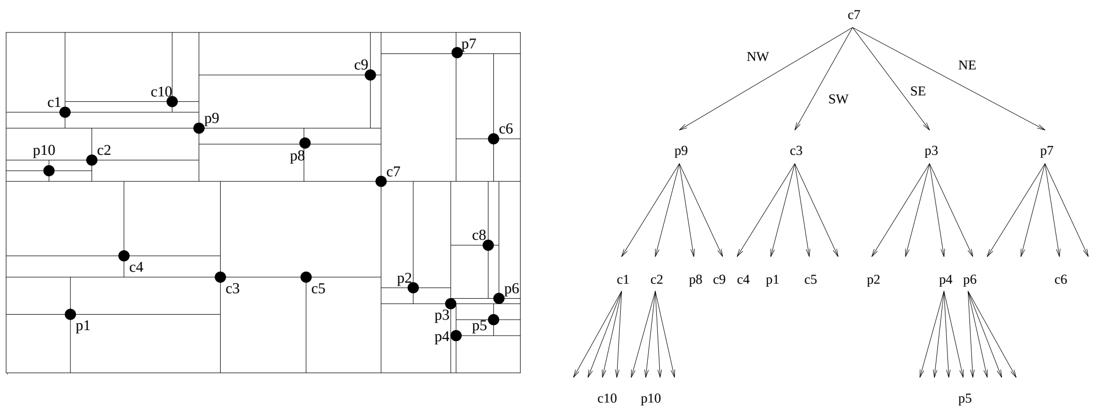


Figure 14: Point Quadtree

Another popular variant is the *region quadtree* (Samet 1984). Region quadtrees are based on a *regular decomposition* of the universe, i.e., the 2^d subspaces resulting from a partition are always of equal size. This greatly facilitates searches. For the running example, Figure 15 shows how region quadtrees can be used to represent sets of points. Here the threshold for the number of points in any given subspace was set to one. In more complex versions of the region quadtree, such as the *PM quadtree* (Samet and Webber 1985), it is also possible to store polygonal data directly. PM quadtrees divide the quadtree regions (and the data objects in them) until they contain only a small number of polygon edges or vertices. These edges or vertices (which together form an exact description of the data objects) are then attached to the leaves of the tree. Another class of quadtree structures has been designed for the management of collections of rectangles; see (Samet 1988) for a survey.

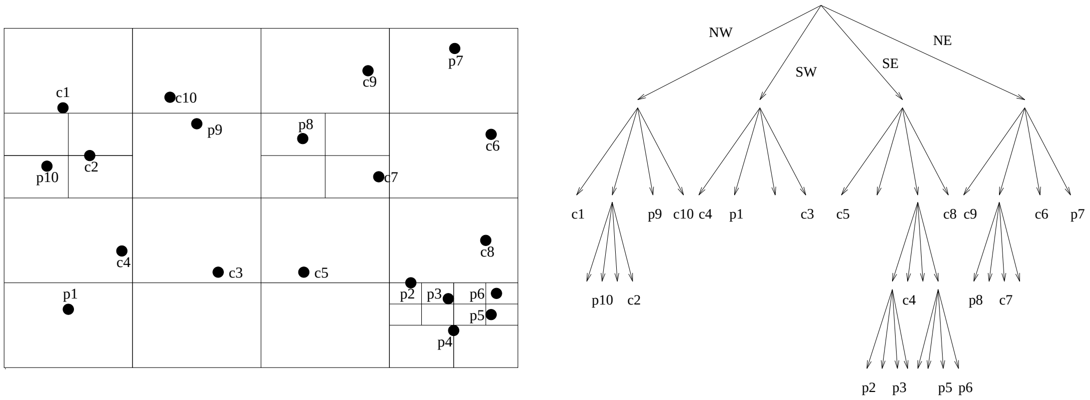


Figure 15: Region Quadtree

4 Point Access Methods

The multidimensional data structures presented in the previous section do not take secondary storage management into account explicitly. They have originally been designed for main memory applications where all the data is available without accessing the disk. Despite growing main memories, this is of course not always the case. In many spatial database applications, such as geography, the amount of data to be managed is notoriously large. While one can certainly use main memory structures for data that resides on disk, their performance will often be considerably below the optimum because there is no control over the way the operating system performs the disk accesses. The access methods presented in this and the following section have been designed with secondary storage management in mind. Their operations are closely coordinated with the operating system to ensure that overall performance is optimized.

As mentioned before, we shall first present a selection of *point access methods (PAMs)*. Usually, the points in the database are organized in a number of buckets, each of which corresponds to a disk page and to some subspace of the universe. The subspaces (often referred to as *data regions*, *bucket regions* or simply *regions*, even though their dimension may be greater than two) need not be rectilinear, although they often are. The buckets are accessed by means of a search tree or some d-dimensional hash function.

The grid file (Nievergelt, Hinterberger, and Sevcik 1984), for example, uses a directory and a grid–like partition of the universe to answer an exact match query with exactly two disk accesses. Furthermore, there are multidimensional hashing schemes (Tamminen 1982; Kriegel and Seeger 1986; Kriegel and Seeger 1988), multilevel grid files (Whang and Krishnamurthy 1985; Hutflesz, Six, and Widmayer 1988b), and hash trees (Ouksel 1985; Otoo 1985), which organize the directory as a tree structure. Tree–based access methods are usually a generalization of the B–tree to higher dimensions, such as the k-d-B-tree (Robinson 1981) or the hB-tree (Lomet and Salzberg 1989).

In the remainder of this section, we first discuss the approaches based on *hashing*, then continue with *hierarchical* (tree-based) access methods. This classification is hardly unambiguous, especially in the presence of an increasing number of hybrid approaches that attempt to combine the advantages of several different techniques. Our approach resembles the classification of Samet (1990) who distinguishes between *hierarchical* methods (point/region quadtrees, k-d-trees, range trees) on the one hand and *bucket* methods (grid file, EXCELL) on the other hand. His discussion of the former is primarily in the context of main memory applications. Our presentation focuses throughout on structures that take secondary storage management into account.

Another interesting taxonomy has been proposed by Seeger and Kriegel (1990) who classify point access methods by the properties of the bucket regions (Table 1). First, they may be pairwise *disjoint* or they may have mutual overlaps. Second, they may have the shape of an *interval* (box) or be of some arbitrary polyhedral shape. Third, they may cover the *complete* universe or just those parts that contain any data objects. This taxonomy results in eight classes, four of which are populated by existing access methods.

property			**point access method**
intervals	complete	disjoint	
×	×	×	quadtree (Finkel and Bentley 1974; Samet 1984), k–d–B tree (Robinson 1981), EXCELL (Tamminen 1982), interpolation hashing (Burkhard 1983), multidimensional extendible hashing (Otoo 1984), grid file (Nievergelt, Hinterberger, and Sevcik 1984), balanced multidimensional two–level grid file (Hinrichs 1985), interpolation–based grid file (Ouksel 1985), extendible hash tree (Otoo 1986), MOLHPE (Kriegel and Seeger 1986), PLOP-hashing (Kriegel and Seeger 1988), quantile hashing (Kriegel and Seeger 1989), LSD-tree (Henrich, Six, and Widmayer 1989)
×	×		twin grid file (Hutflesz, Six, and Widmayer 1988b)
×		×	multilevel grid file (Whang and Krishnamurthy 1985), buddy tree (Seeger and Kriegel 1990)
	×	×	BSP-tree (Fuchs, Kedem, and Naylor 1980), BD-tree (Ohsawa and Sakauchi 1983), BANG file (Freeston 1987), hB–tree (Lomet and Salzberg 1989)

Table 1: Classification of PAMs according to Seeger and Kriegel (1990)

4.1 Multidimensional Hashing

Although there is no total order for objects in two- and higher-dimensional space that completely preserves spatial proximity, there have been numerous attempts to construct hashing functions that preserve proximity at least to some extent. The goal of all these heuristics is that objects that are located close to each other in original space are likely to be stored close together on the disk. This could contribute substantially to minimizing the number of disk accesses per range query. We begin our presentation with several structures based on extendible hashing. Structures based on linear hashing are discussed in Section 4.1.5. The discussion of two hybrid methods, the BANG file and the buddy tree, is postponed until Section 4.2.

4.1.1 The Grid File (Nievergelt, Hinterberger, and Sevcik 1981)

As a typical representative for an access method based on hashing, we will first discuss the *grid file* and some of its variants (Hinrichs 1985; Ouksel 1985; Whang and Krishnamurthy 1985; Six and Widmayer 1988; Blanken et al. 1990). The grid file superimposes a d-dimensional orthogonal *grid* on the universe. Because the grid is not necessarily regular, the resulting *cells* may be of different shapes and sizes. A *grid directory* associates one or more of these cells with *data buckets*, which are stored on one disk page each. Each cell is associated with one bucket, but a bucket may contain several adjacent cells. Since the directory may grow large, it is usually kept on secondary storage. 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 16 shows a grid file for the running example. We assume bucket capacity to be four data points. The center of the figure shows the directory with scales on the x- and y-axes. The data points are displayed in the directory for demonstration purposes only; they are not stored there of course. In the lower left part, four cells are combined into a single bucket, represented by four pointers to a single page. There are thus four directory entries for the same page, which illustrates a well-known problem of the grid file: It suffers from a superlinear growth of the directory even for data that is uniformly distributed (Regnier 1985; Widmayer 1991). The bucket region containing the point c5 could have been merged with one of the neighboring buckets for better storage utilization. Combining such *buddies* may cause problems later, however, when trying to combine buckets that are underoccupied (Hinrichs 1985; Seeger and Kriegel 1990). This is a tradeoff that has to be solved depending on the particular application at hand.

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 main memory, one disk access is necessary. The loaded cell contains a reference to the page where to find possibly matching data. Retrieving this page may require another disk access. Altogether no more than two page accesses are necessary to answer this query. For a range query, one has to examine all cells that overlap the search region. After eliminating duplicates, one fetches the corresponding data pages into memory for a more detailed inspection.

To insert a point, one first performs an exact match query to locate the cell and the data page ν_i where the entry should be inserted. If there is sufficient space left on ν_i, the new entry is inserted. If not, we have to distinguish two cases, depending on the number of grid cells that point to the data page where the new data item is to be

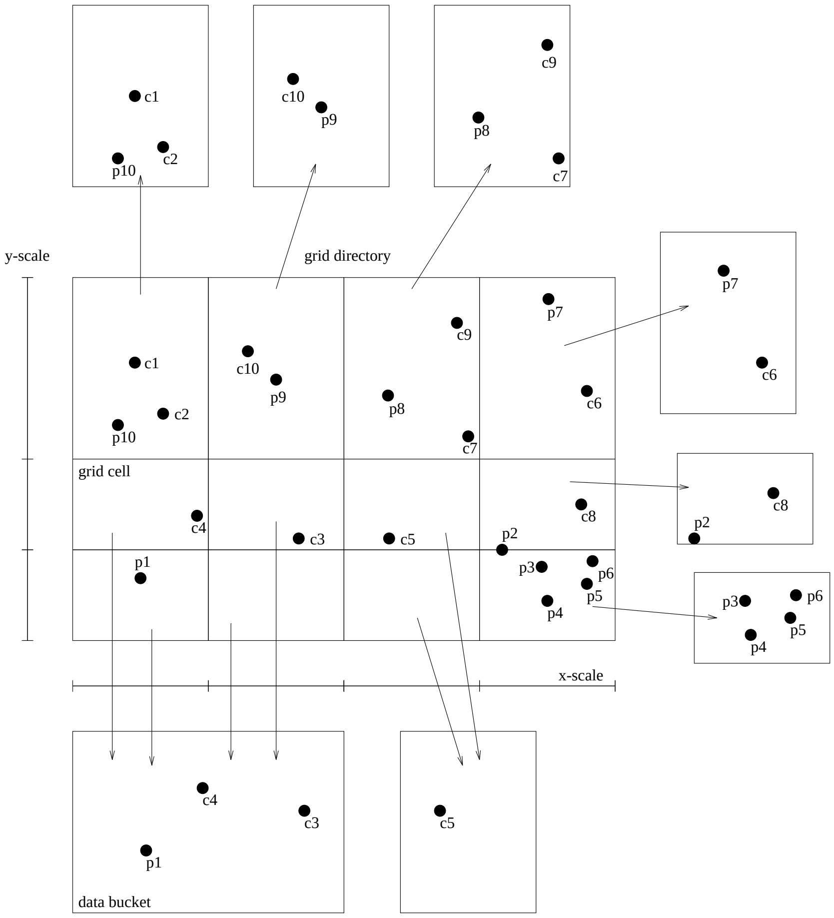


Figure 16: Grid File

inserted. If there are several, one checks whether an existing hyperplane stored in the scales can be used for splitting the data page successfully. If so, a new data page is allocated and the data points are distributed accordingly among the data pages. If none of the existing hyperplanes is suitable, or if only one grid cell points to the data page in question, a splitting hyperplane H is introduced and a new data page ν_j is allocated. The new entry and the entries of the original page ν_i are redistributed among ν_i and ν_j, depending on their location relative to H. H is inserted into the corresponding scale; all cells that intersect H are split accordingly. Splitting is therefore not a local operation and can lead to superlinear directory growth (Regnier 1985; Freeston 1987; Widmayer 1991).

Deletion is not a local operation either. With the deletion of an entry, the storage utilization of the corresponding data page may drop below the given threshold. Depending on the current partitioning of space, it may then be possible to merge this page with a neighbor page and to drop the partitioning hyperplane from the corresponding scale. Depending on the implementation of the grid directory, merging may require a complete directory scan (Hinrichs 1985). Hinrichs discusses several methods for finding candidates with which a given data bucket can merge, including the *neighbor system* and the *multidimensional buddy system*. The neighbor system allows to merge two adjacent regions if the result is a rectangular region again. In the buddy system, two adjacent regions can be merged provided that the joined region can be obtained by a regular binary subdivision of the universe. Both systems are not able to eliminate completely the possibility of a *deadlock*, in which case no merging is feasible because the resulting bucket region would not be box-shaped (Hinrichs 1985; Seeger and Kriegel 1990).

For a theoretical analysis of the grid file and some of its variants see (Regnier 1985) or (Becker 1992). Regnier shows in particular that the average space occupancy of the data buckets is about 69 % (ln 2) for uniformly distributed data.

4.1.2 EXCELL (Tamminen 1982)

Closely related to the grid file is the *EXCELL method* (EXtendible CELL) proposed by Tamminen (1982). 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 halving of *all* cells and therefore in the doubling of the directory size. To alleviate this problem, Tamminen (1983) later suggested a hierarchical method, similar to the multilevel grid file of Whang and Krishnamurthy (1985). Overflow pages are introduced to limit the depth of the hierarchy.

4.1.3 The Two-Level Grid File (Hinrichs 1985)

The basic idea of the *two-level grid file* 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 contain 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. Even though this modification leads to a slower directory growth, it does not solve the problem completely. Furthermore, Hinrichs implicitly assumes that the second level can be kept in main memory, such that the two disk access principle still holds. Figure 17 shows a two-level grid file for the running example. Each cell in the root directory has a pointer to the corresponding entries in the subdirectory, which have their own scales in turn.

4.1.4 The Twin Grid File (Hutflesz, Six, and Widmayer 1988b)

The *twin grid file* tries to increase space utilization compared to the original grid file by introducing a second grid file. As indicated by the name "twin," the relationship between these two grid files is not hierarchical, as in the case of the two-level grid file, but somewhat more balanced. Both grid files span the whole universe. The distribution of the data among the two files is performed dynamically. Hutflesz et al. (1988b) report an average occupancy of 90% for the twin grid file (compared to 69% for the original grid file) without experiencing substantial performance penalties.

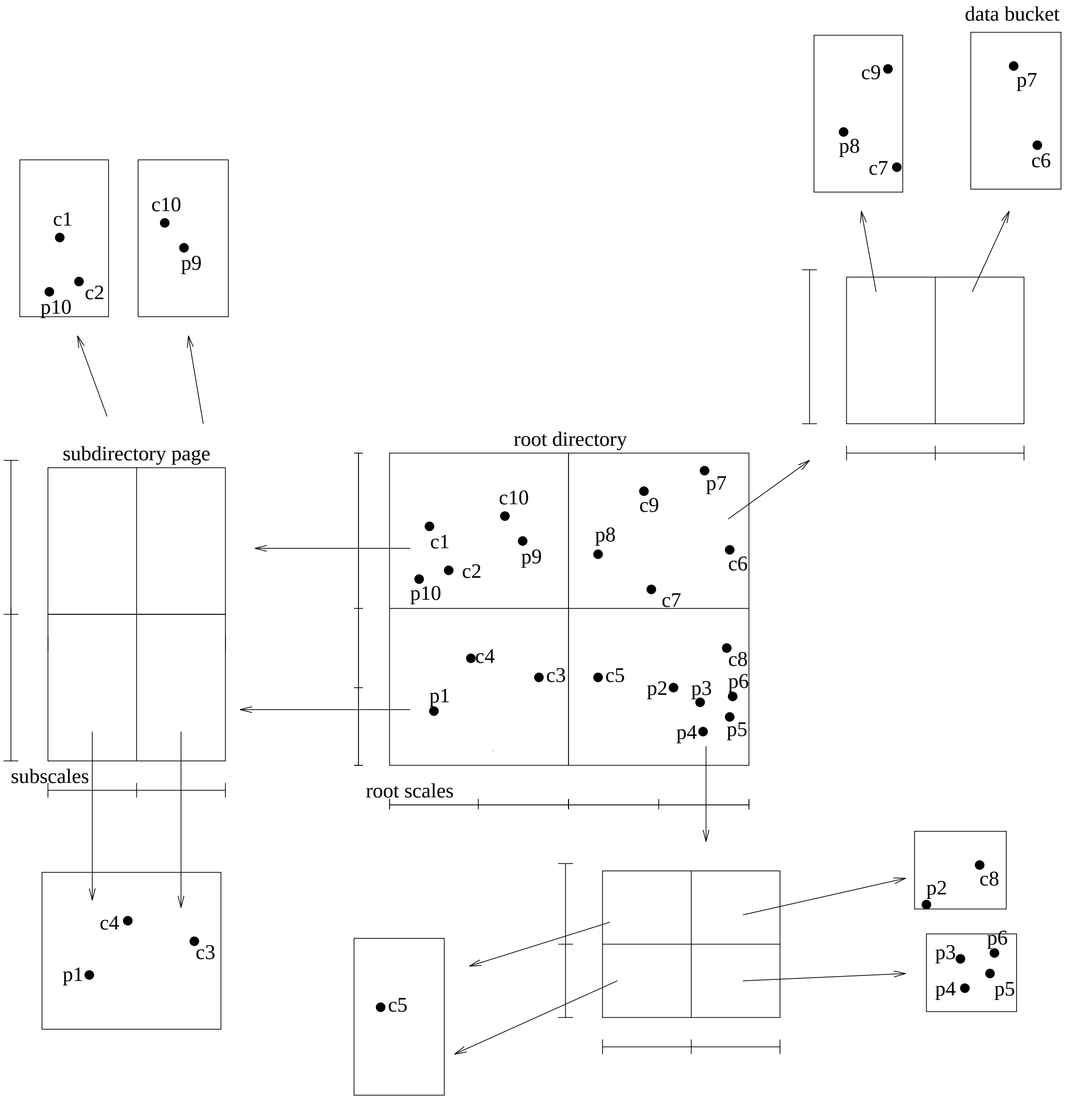


Figure 17: Two-Level Grid File

To illustrate the underlying technique, consider the running example depicted in Figure 18. Let us assume that each bucket can accommodate four points. If the number of points in a bucket exceeds that limit, one possibility is to create a new bucket and redistribute the points among the two new buckets. Before doing so, however, the twin grid file tries 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. Therefore we shift points from P to S if and only if the resulting decrease in the number of buckets in P outweighs the increase in the number of buckets in S. This strategy also favors points to be placed in the primary file in order to form large and empty buckets in the secondary file. Consequently, all points in S can be associated with an empty or a full bucket region of P. Note that there usually exists no unique optimum for the distribution of data points among the two files.

The fact that data points may be found in either of the two grid files requires search operations to visit the two files, which causes some overhead. Nevertheless, the performance results reported by Hutflesz et al. (1988b) indicate that the search efficiency of the twin grid file is competitive with the original grid file. While the twin grid file is somewhat inferior to the original grid file for smaller query ranges, this changes for larger search spaces.

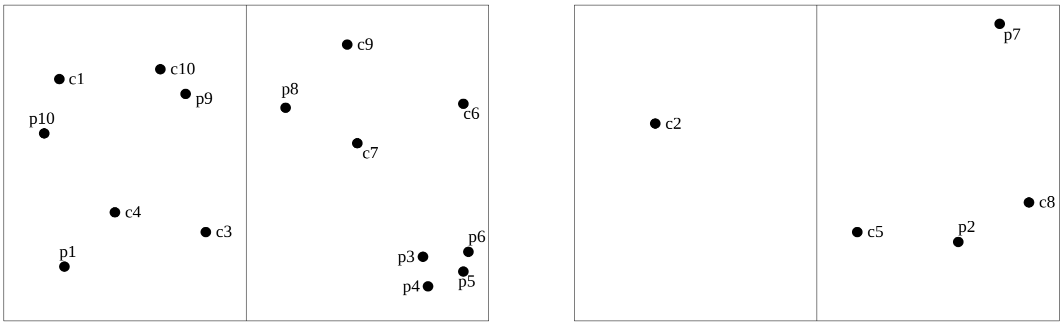


Figure 18: Twin Grid File

4.1.5 Multidimensional Linear Hashing

Unlike multidimensional extendible hashing, multidimensional *linear* hashing uses no or only a very small directory. It therefore occupies relatively little storage compared to extendible hashing, and it is usually possible to keep all relevant information in main memory.

Several different strategies have been proposed to perform the required address computation. While early proposals (Ouksel and Scheuermann 1983) failed to support range queries, Kriegel and Seeger (1986) later proposed a variant of linear hashing called *multidimensional order-preserving linear hashing with partial expansions (MOLHPE)*. This structure is based on the idea of partially extending the buckets without expanding the file size at the same time. To this end, they use a d-dimensional expansion pointer referring to the group of pages that will be expanded next. With this strategy, Kriegel and Seeger can guarantee a modest file growth, at least in the case of well-behaved data. According to their experimental results, MOLHPE outperforms its competitors for uniformly distributed data. It fails, however, for non-uniform distributions, mostly because the hashing function does not adapt gracefully to the given distribution.

To solve this problem, the same authors later applied a stochastic technique (Burkhard 1984) to determine the split points. Because of the name of that technique (α-quantiles), the access method was called *quantile hashing* (Kriegel and Seeger 1987; Kriegel and Seeger 1989). The critical property of the division in quantile hashing is that the original data, which may have a non-uniform distribution, is transformed into uniformly distributed values for α. These values are then used as input to the MOLHPE algorithms for retrieval and update. Since the region boundaries are not necessarily simple binary intervals, a small directory is needed. In exchange, skewed input data can be maintained as efficiently as uniformly distributed data. *PLOP (piecewise linear order-preserving) hashing* has been proposed by the same authors one year later (Kriegel and Seeger 1988). Because this structure can also be used as an access method for extended objects, we delay its discussion until Section 5.2.7.

Another variant that has better order-preserving properties than MOLHPE has been reported by Hutflesz, Six, and Widmayer (1988a). Their *dynamic z-hashing* uses a space-filling technique called *z-ordering* (Orenstein and Merrett 1984) to guarantee that points that are located close to each other are also stored close together on the disk. Z-ordering will be described in detail in Section 5.1.2. One disadvantage of z-hashing is that a number of useless data blocks will be generated, similar as in the interpolation-based grid file (Ouksel 1985). On the other hand, z-hashing allows to read three to four buckets in a row on the average before a seek is required, whereas MOLHPE manages to read only one (Hutflesz et al. 1988a). Widmayer (1991) later noted, however, that both z-hashing and MOLHPE are of limited use in practice, due to their inability to adapt to different data distributions.

4.2 Hierarchical Access Methods

In this section we discuss several PAMs that are based on a binary or multiway tree structure. Except for the BANG file and the buddy tree, which are hybrid structures, they do not perform any address computation. Like hashing-based methods, however, they organize the data points in a number of buckets. Each bucket usually corresponds to a leaf node of the tree (also called *data node*) and a disk page, which contains those points that are located in the corresponding bucket region. The *interior nodes* of the tree (also called *index nodes*) are used to guide the search; each of them typically corresponds to a larger subspace of the universe that contains all bucket regions in the subtree below. A search operation is then performed by a top-down tree traversal.

At this point, individual tree structures still dominate the field although more generic concepts are gradually attracting more attention. The *generalized search tree* by Hellerstein et al. (1995), for example, represents an attempt to subsume many of those common features under a generic architecture.

Differences between individual structures are mainly based on the characteristics of the regions. Table 1 on page 21 showed that in most *PAMs* the regions at the same tree level form a partitioning of the universe, i.e., they are mutually disjoint, with their union being the complete space. For *SAMs* this is not necessarily true; as we will see in Section 5, overlapping regions and partial coverage are important techniques to improve the search performance of SAMs.

4.2.1 The K-D-B-Tree (Robinson 1981)

The *k-d-B-tree* combines some of the properties of the adaptive k-d-tree (Bentley and Friedman 1979) and the B-tree (Comer 1979) to handle multidimensional points. It partitions the universe in the manner of an adaptive k-d-tree and associates the resulting subspaces with tree nodes. Each interior node corresponds to an interval-shaped region. Regions corresponding to nodes at the same tree level are mutually disjoint; their union is the complete universe. The leaf nodes store the data points that are located in the corresponding partition. Like the B-tree, the k-d-B-tree is a perfectly balanced tree that adapts well to the distribution of the data. Other than for B-trees, however, no minimum space utilization can be guaranteed. A k-d-B-tree for the running example is sketched in Figure 19.

Search queries are answered in a straightforward manner, analogously to the k-d-tree algorithms. For the insertion of a new data point, one first performs a point search to locate the right bucket. If it is not full, the entry is inserted. Otherwise, it is split and about half of the entries are shifted to the new data node. In order to find an optimal split, various heuristics are available (Robinson 1981). If the parent index node does not have enough space left to accommodate the new entries, a new page is allocated and the index node is split by a hyperplane. The entries are distributed among the two pages, depending on their position relative to the splitting hyperplane, and the split is propagated up the tree. The split of the index node may also affect regions at *lower* levels of the tree, which have to be split by this hyperplane as well. Because of this *forced split* effect, it is not possible to guarantee a minimum storage utilization.

Deletion is straightforward. After performing an exact match query, the entry is removed. If the number of entries drops below a given threshold, the data node may be merged with a sibling data node as long as the union remains a d-dimensional interval. The procedure to find a suitable sibling node to merge with may involve several nodes. The union of data pages results in the deletion of at least one hyperplane in the parent index node. If an underflow occurs, the deletion has to be propagated up the tree.

4.2.2 The LSD-Tree (Henrich, Six, and Widmayer 1989)

We list the *LSD (Local Split Decision) tree* as a point access method although its inventors emphasize that the structure can also be used for managing extended objects. This claim is based on the fact that the LSD-tree adapts well to data that is non-uniformly distributed and that it is therefore well-suited for using it in connection with the transformation technique; a more detailed discussion of this approach follows in Section 5.1.1.

The directory of the LSD-tree is organized as an adaptive k-d-tree, partitioning the universe into disjoint cells of various sizes. In comparison to the fixed binary partitioning, this results in a better adaption to the data distribution. While the k-d-tree may be arbitrarily unbalanced, the LSD-tree preserves the external balancing property, i.e., the height of its external subtrees differ at most by one. This property is maintained by a special paging algorithm. If the structure becomes too large to fit in main memory, this algorithm identifies subtrees that can be paged out such that the external balancing property is preserved. While efficient, this special paging strategy is obviously a major impediment for the integration of the LSD-tree into a general-purpose database system. Figure 20 shows an LSD-tree for the running example with one external directory page.

As indicated above, the split strategy of the LSD-tree does not assume the data

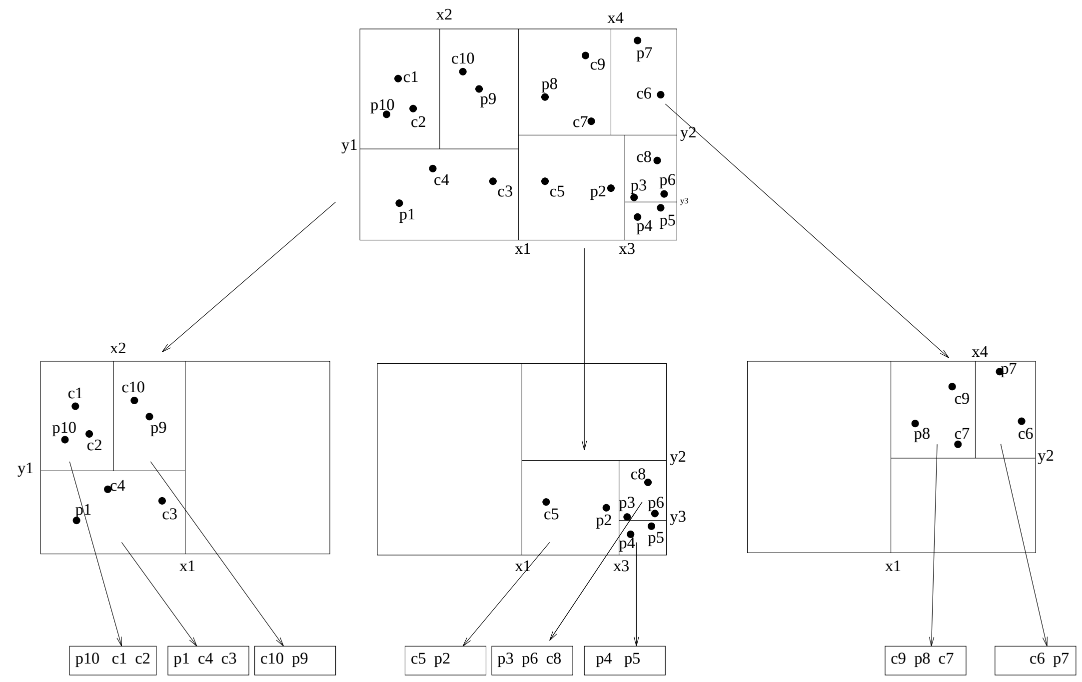


Figure 19: K-D-B-Tree

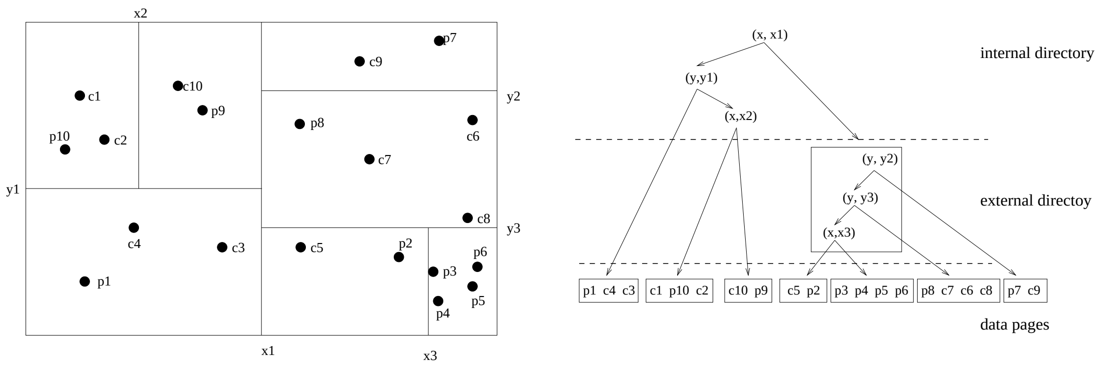


Figure 20: LSD-Tree

to be uniformly distributed. On the contrary, it tries to accommodate skewed data by combining two split strategies:

- *data dependent* (SP_1):
  The choice of the split depends on the data and tries to achieve a most balanced structure, i.e., there should be an equal number of objects on both sides of the split. As the name of the structure suggests, this split decision is made locally.

- *distribution dependent* (SP_2):
  The split is done at a fixed dimension and position. The given data is not taken into account because an underlying (known) distribution is assumed.

To determine the split position SP, one computes the linear combination of the split locations that would result from applying just one of those strategies:

$$SP = \alpha SP_1 + (1 - \alpha) SP_2$$

The factor α is determined empirically based on the given data; it can vary as objects are inserted and deleted from the tree.

Henrich (1995) later presented two algorithms to improve the storage utilization of the LSD-tree by redistributing data entries among buckets. Since these strategies make the LSD-tree sensitive to the insertion sequence, the splitting strategy must be adapted accordingly. In order to improve the search performance for non-point data and range queries, Henrich and Möller (1995) suggest to store auxiliary information on the existing data regions along with the index entries of the LSD-tree.

4.2.3 The Buddy Tree (Seeger and Kriegel 1990)

The buddy tree is a dynamic hashing scheme with a tree-structured directory. The tree is constructed by consecutive insertion, cutting the universe recursively into two parts of equal size with iso-oriented hyperplanes. Each interior node ν corresponds to a d-dimensional partition $P^d(\nu)$ and to an interval $I^d(\nu) \subseteq P^d(\nu)$. $I^d(\nu)$ is the MBB of the points or intervals below ν. Partitions P^d (and therefore intervals I^d) that correspond to nodes on the same tree level are mutually disjoint. As in all tree-based structures, the leaves of the directory point to the data pages. Other important properties of the buddy tree include:

1. Each directory node contains at least two entries.
2. Whenever a node ν is split, the MBBs $I^d(\nu_i)$ and $I^d(\nu_j)$ of the two resulting subnodes ν_i and ν_j are recomputed to reflect the current situation.
3. Except for the root of the directory, there is exactly one pointer referring to each directory page.

Due to property 1, the buddy tree may not be balanced, i.e., the leaves of the directory may be on different levels. Property 2 tries to achieve a high selectivity at the directory level. Properties 1 and 3 make sure that the growth of the directory remains linear. To avoid the deadlock problem of the grid file, the buddy tree uses k-d-tries (Orenstein 1982) to partition the universe. Only a restricted number of buddies are admitted, namely those that could have been obtained by some recursive halving of the universe. However, as shown by Seeger and Kriegel (1990), the number of possible buddies is larger than in the grid file and other structures, which makes the buddy tree more flexible in the case of updates. Experiments by Kriegel et al. (1990) indicate that the buddy tree is superior to several other PAMs, including the hB-tree, the BANG file, and the two-level grid file. A buddy tree for the running example is shown in Figure 21.

Two older structures, the *interpolation-based grid file* by Ouksel (1985) and the *balanced multidimensional extendible hash tree* by Otoo (1986), are both special cases

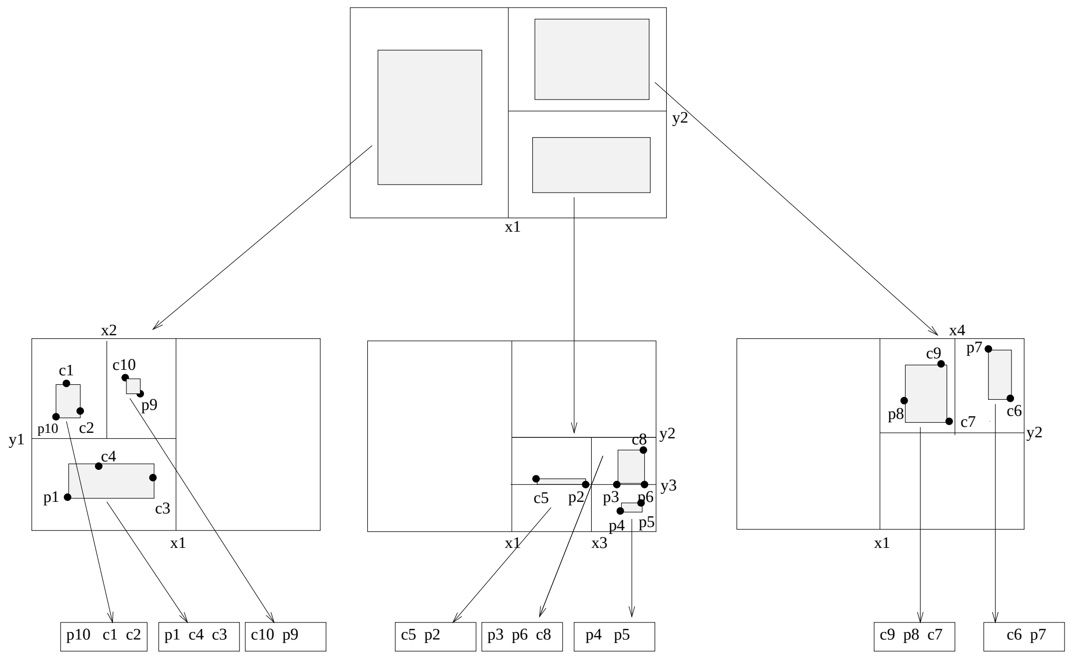


Figure 21: Buddy Tree

of the buddy tree. They can be obtained by restricting the properties of the regions. Interpolation-based grid files avoid the excessive growth of the grid file directory by representing blocks explicitly, which guarantees that there is only one directory entry for each data bucket. The disadvantage of this approach is that empty regions have to be introduced in the case of skewed data input. Seeger (1991) later showed that the buddy tree can easily be modified to handle spatially extended objects by using one of the techniques presented in Section 5.

4.2.4 The BANG File (Freeston 1987)

To obtain a better adaption to the given data points, Freeston (1987) proposed a new structure, which he called *BANG (Balanced And Nested Grid) file* - even though it differs from the grid file in many aspects. Similar to the grid file, it partitions the universe into intervals (boxes). What is different, however, is that in the BANG file bucket regions may intersect, which is not possible in the regular grid file. In particular, one can form non-rectangular bucket regions by taking the geometric difference of two or more intervals *(nesting)*. To increase storage utilization, it is possible during insertion to redistribute points between different buckets. To manage the directory, the BANG file uses a balanced search tree structure. In combination with the hash-based partitioning of the universe, the BANG file can therefore be viewed as a hybrid structure.

Figure 22 shows the BANG file for the running example. Three rectangles have been cut out of the universe R1: R2, R5, and R6. In turn, the rectangles R3 and R4 are nested into R2 and R5, respectively. If one represents the resulting space partitioning as a tree using bit interleaving, one obtains the structure shown on the right hand side of Figure 22. Here the asterisk represents the empty string, i.e., the universe. A comparison with Figure 13 shows that the BANG file can in fact be regarded as a paginated version of the BD-tree discussed in Section 3.2.3.

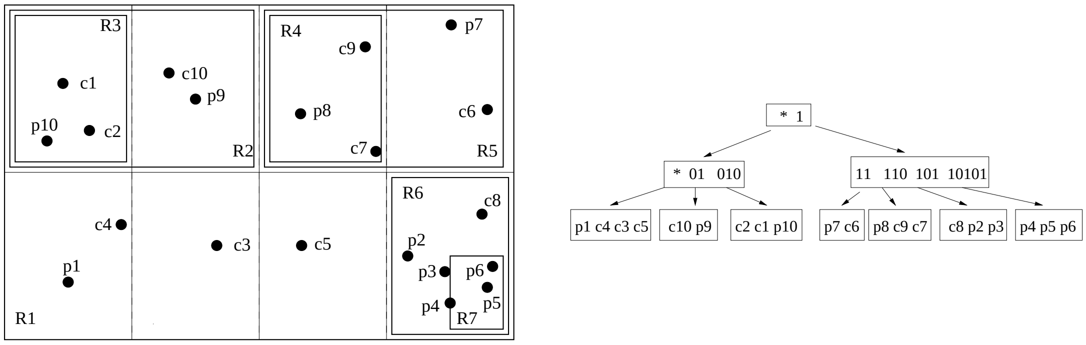


Figure 22: BANG File

In order to achieve a high storage utilization, the BANG file performs spanning splits which may lead to the displacement of parts of tree. As a result, a point search may in the worst case require the traversal of the entire directory in a depth-first manner. To address this problem, Freeston (1989a) later proposed different splitting strategies which avoid the spanning problem at the expense of a potentially low storage utilization. Kumar (1994a) proposed the same idea starting from the BD-tree and called the resulting structure *G-tree* (*grid tree*) (Kumar 1994a). With regard to the BD-tree, the structure differs in the way the partitions are mapped into buckets. To obtain a simpler mapping, the G-tree sacrifices the minimum storage utilization that holds for the BD-tree.

While the data partitioning given in Figure 22 is feasible for the BD-tree and the original BANG file, it can not be achieved with the BANG file using forced splits (Freeston 1989a). For this variant, we would have to split the root and move, for example, entry c5 to the bucket containing the entries p7 and c6.

Freeston (1989b) also proposed an extension to the BANG file to handle extended objects. As often found in PAM extensions, the centroid is used to determine the bucket where to place a given object. To take account for the object's spatial extension, the bucket regions are extended where necessary (Seeger and Kriegel 1988; Ooi 1990).

Ouksel and Mayer (1992) later proposed an access method called *nested interpolation-based grid file*, which is closely related to the BANG file. The major difference concerns the way the directory is organized. In essence, the directory consists of a list of one-dimensional access methods (e.g., B-trees) storing the z-order encoding of the different data regions, along with pointers to the respective data buckets. By doing so, Ouksel and Mayer improved the worst-case bounds from linear (as in the case of the BANG file) to logarithmic.

4.2.5 The hB-Tree (Lomet and Salzberg 1989; Lomet and Salzberg 1990)

The *hB-tree* (holey brick tree) is related to the k-d-B-tree in that it utilizes k-d-trees to organize the space represented by its interior nodes. One of the most noteworthy differences is that node splitting is based on multiple attributes. As a result, nodes no longer correspond to d-dimensional intervals, but to intervals from which smaller intervals have been excised. Similar to the BANG file, the result is a somewhat fractal structure (a *holey brick*) with an external *enclosing region* and several cavities called *extracted regions*. As we will see later, this technique avoids the cascading of splits that is typical for many other structures.

In order to minimize redundancy, the k-d-tree corresponding to an interior node can have several leaves pointing to the same child node. Strictly speaking, the hB-tree is therefore no longer a tree but a directed acyclic graph. With regard to the geometry, this corresponds to the union of the corresponding regions. Once again, the resulting region is typically no longer box-shaped. This peculiarity is illustrated in Figure 23, which shows an hB-tree for the running example. Here the root node contains two pointers to its left descendant node. Its corresponding region u is the union of two rectangles: the one to the left of $x1$ and the one above $y1$. The remaining space (the right lower quadrant) is excluded from u, which is made explicit by the entry *ext* in the corresponding k-d-tree. A similar observation applies to region G, which is again L-shaped: it corresponds to the NW, the SE, and the NE quadrant of the rectangle above $y1$.

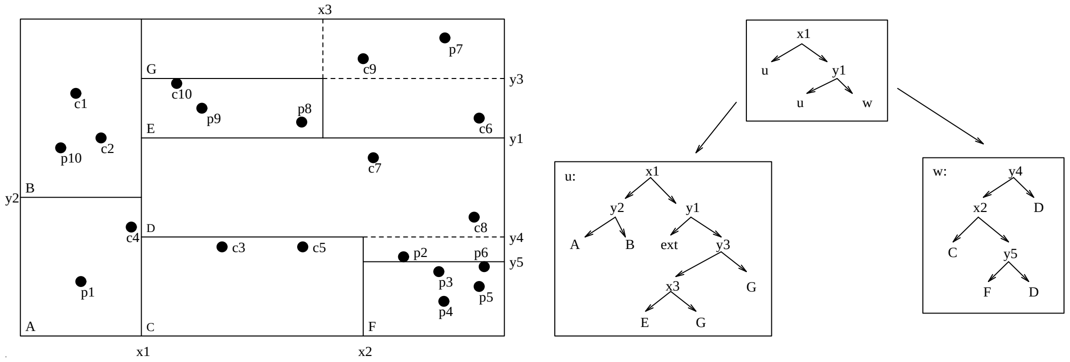


Figure 23: hB-Tree

Searching is similar to the k-d-B-tree; each internal k-d-tree is traversed as usual. Insertions are also carried out analogously to the k-d-B-tree until a leaf node reaches its capacity and a split is required. Instead of using just one single hyperplane to split the node, the hB-tree split is based on more than one attribute and on the internal k-d-tree of the data node to be split. Lomet and Salzberg (1989) show that this policy guarantees a worst-case data distribution between the two resulting two nodes of 1/3 : 2/3. This observation is not restricted to the hB-tree but generalizes to other access methods such as the BD-tree and the BANG file.

The split of the leaf node causes the introduction of an additional k-d-tree node to describe the resulting subspace. This may in turn lead to the split of the ancestor node and its k-d-tree. Since k-d-trees are not height-balanced, splitting the tree at its root may lead to an unbalanced distribution of the nodes. The tree is therefore usually split at a lower level, which corresponds to the excision of a convex region from the space corresponding to the node to be split. The entries belonging to that subspace are extracted and moved to a new hB-tree node. To reflect the absence of the excised region, the hB-tree node is assigned an *external marker*, which indicates that the region is no more a simple interval. With this technique the problem of forced splits is avoided. Splits are local and do not have to be propagated downwards.

In summary, the leaf nodes of the internal k-d-trees are used to

- reference a collection of data records;
- reference other hB-tree nodes;
- indicate that a part of this tree has been extracted.

In a later Ph.D. thesis (Evangelidis 1994), the hB-tree has been extended to allow for concurrency and recovery by modifying the hB-tree in such a way that it becomes a special case of the Π-tree (Lomet and Salzberg 1992). Consequently, the new structure is called hB$^{\Pi}$-*tree* (Evangelidis, Lomet, and Salzberg 1995). As a result of these modifications, the new structure can immediately take advantage of the Π-tree node consolidation algorithm. The lack of such an algorithm has been one of the major weaknesses of the hB-tree. Furthermore, the hB$^{\Pi}$-tree corrects a flaw in the splitting/posting algorithm of the hB-tree that may occur for more than three index levels. The essential idea of the correction is to impose restrictions on the splitting/posting algorithms, which in turn affects the space occupancy.

One minor problem remains: As mentioned, the hB-tree may store several references to the same child node. While the number of nodes may in principle expose a growth behavior that is superlinear in the number of regions, this observation seems of mainly theoretical interest. According to the authors of the hB$^{\Pi}$-tree (Evangelidis, Lomet, and Salzberg 1995), it is quite rare that more than one leaf of the underlying k-d tree refers to any given child. In their experiments, more than 95% of the index nodes and all of the data nodes had only one such reference.

4.2.6 The BV-Tree (Freeston 1995)

The *BV-tree* represents an attempt to solve the d-dimensional B-tree problem, i.e., to find a generic generalization of the B-tree to higher dimensions. The BV-tree is not meant to be a concrete access method. It represents a conceptual framework that can be applied to a variety of existing access methods, including the BANG file or the hB-tree.

Freeston's proposal is based on the conjecture that one can maintain the major strengths of the B-tree in higher dimensions, provided one relaxes the strict requirements concerning tree balance and storage utilization. The BV-tree is not completely balanced. Furthermore, while the B-tree guarantees a worst-case storage utilization of 50%, Freeston argues that such a comparatively high storage utilization cannot be ensured for higher dimensions for topological reasons. However, the BV-tree manages to achieve the 33% lower bound suggested by Lomet and Salzberg (1989).

To achieve a guaranteed worst-case search performance, the BV-tree combines the excision concept (Freeston 1987) with a technique called *promotion*. Here, intervals from lower levels of the tree are moved up the tree, i.e., closer to the root. To keep track of the resulting changes, with each promoted region we store a level number (called a *guard*) that denotes the region's original level.

The search algorithms are based on a notional backtracking technique. While descending the tree, we store possible alternatives (relevant guards of the different index levels) in a *guard set*. The entries of this set act as backtracking points and represent a single path from the root to the level currently inspected; for point queries, they can be maintained as a stack. To answer a point query, we start at the root and inspect all node entries whether the corresponding regions overlap the search point. Among those entries inspected, we choose the best-matching entry to investigate next. Possibly we also store some guards in the guard set. At the next level this procedure is repeated recursively, this time taking the stored guards into account. Before following the best-matching entry down to the next level, the guard set is updated by merging the matching new guards with the existing ones. Two guards at the same level are merged by discarding the poorer match. This search continues recursively until we reach the leaf level. Note that for point queries, the length of the search path is equal to the height of the BV-tree because each region in space is represented by a unique node entry.

Figure 24 shows a BV-tree and the corresponding space partitioning for the running example. For illustration purposes we do not confine the grouped regions or objects by a tight polyline, but by a loosely wrapped boundary. In this example, the region D0 acts as a guard. It is clear from the space partitioning that D0 originally belongs to the bottom index level (that is, the middle level in the figure). Since it functions as a guard for the enclosed region S1, however, it has been promoted to the root level. Suppose we are interested in all objects intersecting the black rectangle X. Starting at the root, we place D0 in the guard set and investigate S1. Because the inspection of S1 reveals that the search region is neither included in P0, nor in N0 or M0, we backtrack to D0 and inspect the entries for D0. In our example, no entry satisfies the query.

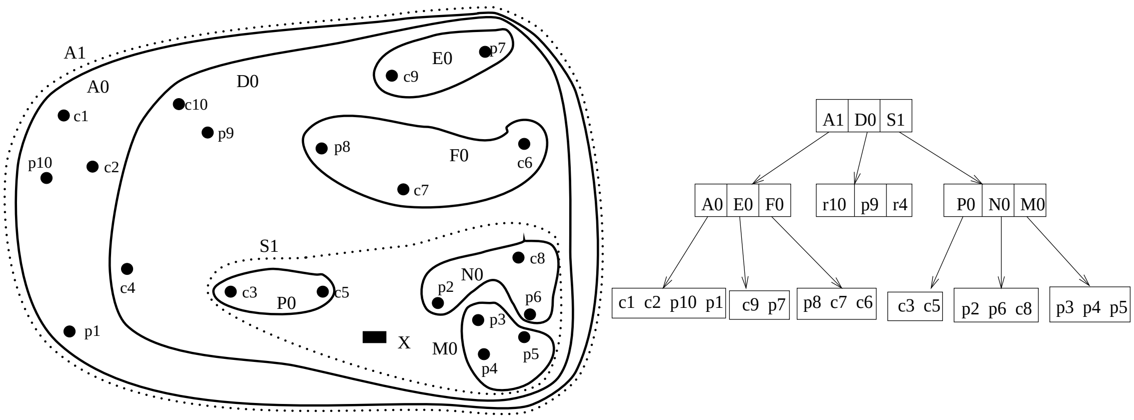


Figure 24: BV-Tree

In a later paper, Freeston (1997) discusses complexity issues related to updates of guards. In the presence of such updates, it is necessary to "downgrade" (demote) entries which are no longer guards, which may in turn affect the overall structure negatively. Freeston's conclusion is that the logarithmic access performance and the minimum storage utilization of the BV-tree can be preserved by postponing the demotion of such entries, which may lead to (very) large index nodes.

4.3 Space-Filling Curves for Point Data

We already mentioned the main reason why the design of multidimensional access methods is so difficult compared to the one-dimensional case: There is no total order that preserves spatial proximity. One way out of this dilemma is to find heuristic solutions, i.e., to look for total orders that preserve spatial proximity at least to some extent. The idea is that if two objects are located close together in original space, there should *at least be a high probability* that they are close together in the total order, i.e., in the one-dimensional image space. For the organization of this total order one could then use a one-dimensional access method (such as a B^+-tree), which may provide good performance at least for some spatial queries.

Research on the underlying mapping problem goes back well into the last century; see (Sagan 1994) for a survey. With regard to its relevance for spatial searching, Samet (1989) provides a good overview of the subject. One thing all proposals have in common is that they first partition the universe with a grid. Each of the grid cells is labeled with a unique number that defines its position in the total order (the *space-filling curve*). The points in the given data set are then sorted and indexed according to the grid cell they are contained in. Note that while the labeling is independent of the given data, it is obviously critical for the preservation of proximity in one-dimensional address space. That is, the way we label the cells determines how clustered adjacent cells are stored on secondary memory.

Figure 25 shows four common labelings. Figure 25a corresponds to a row-wise enumeration of the cells (Samet 1989). Figure 25b shows the cell enumeration imposed by the *Peano curve* (Morton 1966), also called *quad codes* (Finkel and Bentley 1974), *N-trees* (White 1981), *locational codes* (Abel and Smith 1983), or *z-ordering* (Orenstein and Merrett 1984). Figure 25c shows the *Hilbert curve* (Faloutsos and Roseman 1989; Jagadish 1990a), and Figure 25d depicts *Gray ordering* (Faloutsos 1986; Faloutsos 1988), which is obtained by interleaving the Gray codes of the x- and y-coordinates in a bitwise manner. Gray codes of successive cells differ in exactly one bit.

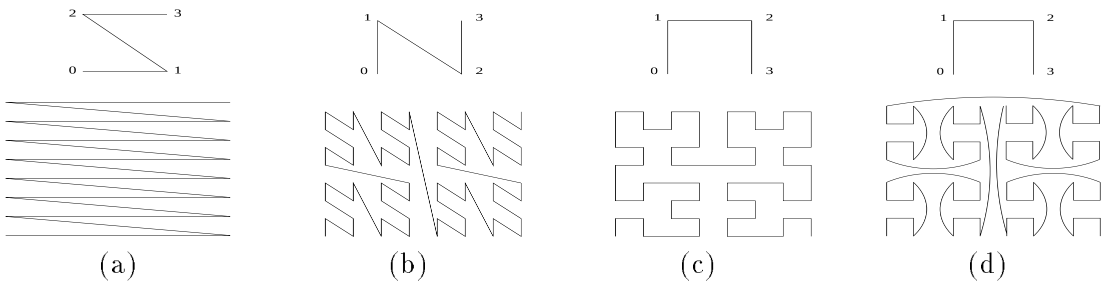


Figure 25: Four Space-Filling Curves

Based on several experiments, Abel and Mark (1990) conclude that z-ordering and the Hilbert curve are most suitable as a multidimensional access method. Jagadish (1990a) and Faloutsos and Rong (1991) all prefer the Hilbert curve among those two.

Z-ordering is one of the few access methods that has found its way into commercial database products. In particular, Oracle has adapted and integrated the technique into its database system (Oracle Inc. 1995).

An important advantage of all space-filling curves is that they are practically insensitive to the number of dimensions if the one-dimensional keys can be arbitrarily large. Everything is mapped into one-dimensional space, and one's favorite one-dimensional access method can be applied to manage the data. An obvious disadvantage of space-filling curves is that incompatible index partitions cannot be joined without recomputing the codes of at least one of the two indexes.

5 Spatial Access Methods

All multidimensional access methods presented in the previous section have been designed to handle sets of data points and to support spatial searches on them. None of those methods is directly applicable to databases containing objects with a spatial extension. Typical examples include geographic databases, containing mostly polygons, or mechanical CAD data, consisting of three-dimensional polyhedra. In order to handle such extended objects, point access methods have been modified using one of the following four techniques:

1. Transformation (Object Mapping)
2. Overlapping Regions (Object Bounding)
3. Clipping (Object Duplication)
4. Multiple Layers

A simpler version of this classification was first introduced by Seeger and Kriegel (1988). Later on, Kriegel et al. (1991) added another dimension to this taxonomy: a spatial access method's *base type*, i.e., the spatial data type it supports primarily. Table 2 shows the resulting classification of spatial access methods. Note that most structures use the interval as a base type.

In the following sections, we present each of those four techniques in some more detail, together with several SAMs based on it.

5.1 Transformation

One-dimensional access methods (Section 3.1) and PAMs (Section 4) can often be used to manage spatially extended objects, provided the objects are first transformed into a different representation. There are essentially two options: one can either transform each object into a higher-dimensional point (Hinrichs 1985; Seeger and Kriegel 1988), or transform it into a set of one-dimensional intervals by means of space-filling curves. We discuss the two techniques in turn.

technique	**base type**			
	grid cell	interval (box)	sphere	polyhedron
transfor-mation	zkdB^+-tree (Orenstein 1986), BANG file (Freeston 1987), hB-tree (Lomet and Salzberg 1989)	all PAMs described in Section 4 except of the BANG file and the hB-tree		P-tree (Jagadish 1990c)
overlapping regions		R-tree (Guttman 1984), R*-tree (Beckmann et al. 1990), skd-tree (Ooi et al. 1987), GBD-tree (Ohsawa and Sakauchi 1990), Hilbert R-tree (Kamel and Faloutsos 1994), buddy tree with overlapping (Seeger 1991)	sphere tree (Oosterom 1990)	P-tree (Schiwietz 1993), KD2B-tree (Oosterom 1990)
clipping		EXCELL (Tamminen 1982), extended k-d-tree (Matsuyama et al. 1984), R^+-tree (Sellis et al. 1987), buddy tree with clipping (Seeger 1991)		cell tree (Günther 1988)
multiple layers		multi-layer grid file (Six and Widmayer 1988), R-file (Hutflesz et al. 1990)		

Table 2: Classification of SAMs Following (Kriegel et al. 1991)

5.1.1 Mapping to Higher-Dimensional Space

Simple geometric shapes can be represented as *points in higher-dimensional space*. For example, it takes four real numbers to represent a (two-dimensional) rectangle in E^2. Those numbers may be interpreted as coordinates of a point in E^4. One possibility is to take the x- and y-coordinates of two diagonal corners *(endpoint transformation)*, another option is based on the centroid and two additional parameters for the extension of the object in x- and y-direction *(midpoint transformation)*. Any such transformation maps a database of rectangles onto a database of 4-dimensional points, which can then be managed by one of the PAMs discussed in the previous section. Search operations can be expressed as point and region queries in this dual space.

If the original database contains more complex objects, they have to be approximated - e.g. by a rectangle or a sphere - before transformation. In this case, the point access method can only lead to a partial solution (cf. Figure 2 on page 9).

Figure 26 shows the dual space equivalents of some common queries. For presentation purposes, the figure shows a mapping from intervals in E^1 to points in E^2. The upper part depicts the results for the endpoint transformation, the lower for the midpoint transformation. Figure 26a shows the transformation result for the range query with the search range $[l, u]$. In dual space this range query maps into a general region query. Any point in dual space that lies in the shaded area corresponds to an interval in original space that overlaps the search interval $[l, u]$, and vice versa. Enclosure and containment queries with the interval $[l, u]$ as argument also map into general region queries (Figure 26b). A point query, finally, maps into a range query for the endpoint transformation. If the midpoint transformation has been employed, however, the point query translates into a general region query as well. Figure 26c gives an example with p as the (one-dimensional) query point.

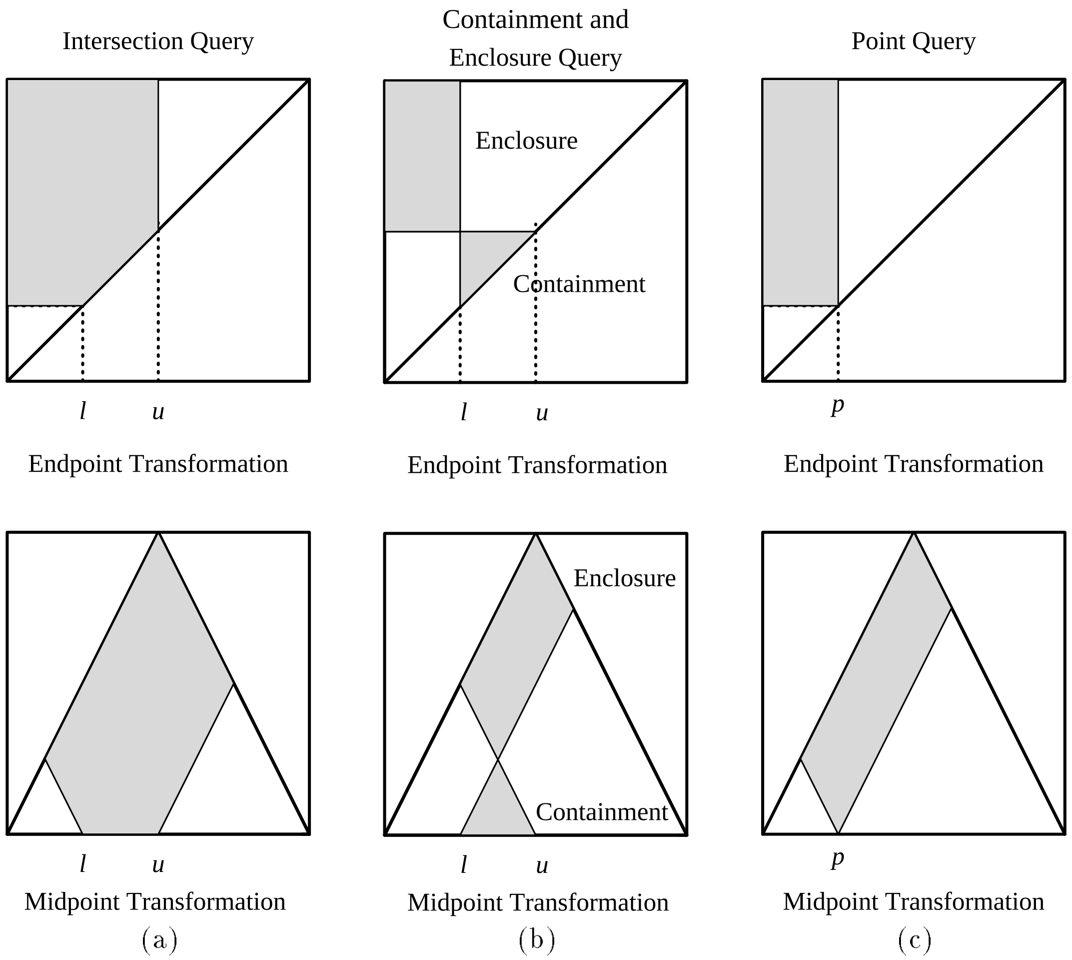


Figure 26: Search Queries in Dual Space

Notwithstanding its conceptual elegance, this approach has several disadvantages. First, as the examples above already indicate, the formulation of point and range queries in dual space is usually much more complicated than in original space (Nievergelt and Hinrichs 1987). Finite search regions may map into infinite search regions in dual space, and some more complex queries involving spatial predicates may not be expressible at all anymore (Henrich et al. 1989; Orenstein 1990; Pagel et al. 1993). Second, depending on the chosen mapping, the distribution of points in dual space may be highly non-uniform even though the original data is uniformly distributed. With the endpoint transformation, for example, there are no image points below the main diagonal (Faloutsos et al. 1987). Third, the images of two objects that are close in original space may be arbitrarily far apart from each other in dual space.

To overcome some of these problems, Henrich et al. (1989), Faloutsos and Rong (1991), as well as Pagel et al. (1993) have proposed special transformation and split strategies. A structure that was designed explicitly to be used in connection with the transformation technique is the LSD-tree (cf. Section 4.2.2). Performance studies by Henrich and Six (1991) confirm the claim that the LSD-tree adapts well to non-uniform distributions, which is of particular relevance in this context. It also contains a mechanism to avoid searching large empty query spaces, which may occur as a result of the transformation.

5.1.2 Space-Filling Curves for Extended Objects

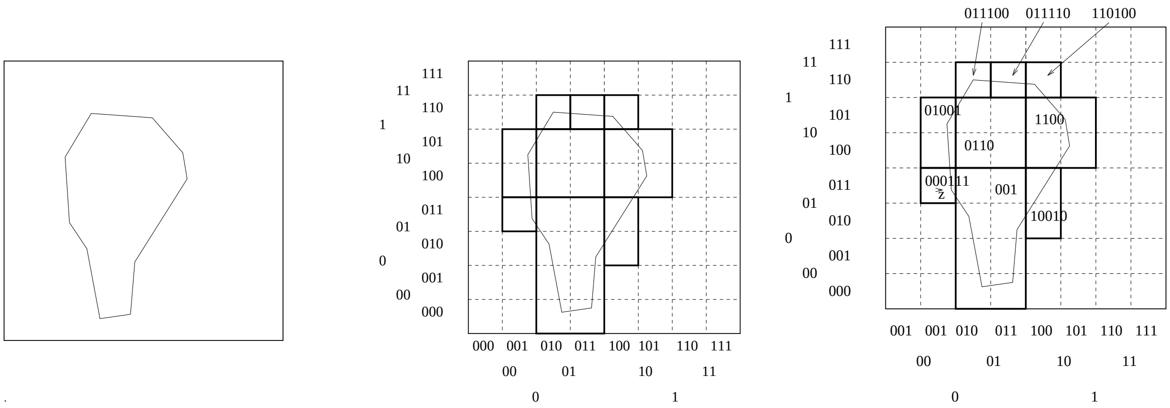
Space-filling curves (cf. Section 4.3) are a very different type of transformation approach that seems to suffer less from some of the drawbacks listed in the previous section. Space-filling curves can be used to represent extended objects by a list of grid cells or, equivalently, a list of one-dimensional intervals that define the position of the grid cells concerned. A In other words, a complex spatial object is approximated not by only *one* simpler object, but by the union of *several* such objects.

The z-ordering approach by Orenstein and Merrett (1984) is one of the more popular approaches of this kind. A simple algorithm to obtain the z-ordering representation of a given extended object can be described as follows. Starting from the (fixed) universe containing the data object, space is split recursively into two subspaces of equal size by $(d-1)$-dimensional hyperplanes. As in the k-d-tree, the splitting hyperplanes are iso-oriented, and their directions alternate in fixed order among the d possibilities. The subdivision continues until one of the following three conditions holds:

1. The current subspace does not overlap the data object.
2. The current subspace is fully enclosed in the data object.
3. Some given level of accuracy has been reached.

The data object is thus represented by a set of cells, called *Peano regions* or *z-regions*. As shown in Section 3.2.3, each such Peano region can be represented by a unique bit string, called *Peano code*, *ST_MortonNumber*, *z-value*, or *DZ-expression*. Using those bit strings, the cells can then be stored in a standard one-dimensional index, such as a B$^+$-tree.

Figure 27 shows a simple example. Figure 27a shows the polygon to be approximated, with the frame representing the universe. After several splits, starting with a vertical split line, we obtain Figure 27b. Nine Peano regions of different shapes and sizes approximate the object. The labeling of each Peano region is shown in Figure 27c. As an example consider the Peano region $\overline{z}$ in the lower left part of the given polygon. It lies to the left of the first vertical hyperplane and below the first horizontal hyperplane, resulting in the first two bits being `00`. As we further partition the lower left quadrant, $\overline{z}$ lies on the left of the second vertical hyperplane, but above the second horizontal hyperplane. The complete bit string accumulated so far is therefore `0001`. In the next round of decompositions, $\overline{z}$ lies to the right of the third vertical hyperplane and above the third horizontal hyperplane, resulting in two additional `1`'s. The complete bit string describing $\overline{z}$ is therefore `000111`.



(a) (b) (c)

Figure 27: Z-Ordering of a Polygon

Figures 27b and 27c also give some bit strings along the coordinate axes, which describe only the splits orthogonal to the given axis. The string `01` on the x-axis, for example, describes the subspace that lies to the left of the first vertical split and to the right of the second vertical split. By bit-interleaving the bit strings that one finds when projecting a Peano region onto the coordinate axes, we obtain its Peano code. Note that if a Peano code z_1 is the prefix of some other Peano code z_2, the Peano region corresponding to z_1 *encloses* the Peano region corresponding to z_2. The Peano region corresponding to `00`, for example, encloses the regions corresponding to `0001` and `000`. Leading `0`'s are therefore significant. This is an important observation, since it can be used for query processing (Gaede and Riekert 1994).

As z-ordering is based on an underlying grid, the resulting set of Peano regions is usually only an approximation of the original object. The termination criterion depends on the accuracy or *granularity* (maximum number of bits) desired. More Peano regions obviously yield more accuracy, but they also increase the overhead, which affects the overall performance of the resulting data structure. As pointed out by Orenstein (1989b) there are two possibly conflicting objectives: First, the number of Peano regions to approximate the object should be small, since this results in less index entries. Second, the accuracy of the approximation should be high, since this reduces the expected number of false drops (i.e., objects that are paged in from secondary memory, only to find out that they do not satisfy the search predicate). For a detailed discussion of this problem, see (Orenstein 1989a), (Orenstein 1989b), and (Gaede 1995b). By enhancing the z-ordering encoding with a single bit that reflects for each Peano region whether it is enclosed in the extended object or not, it is possible to improve the performance of z-ordering even further (Gaede 1995a). Figure 28 shows Peano regions for the running example.

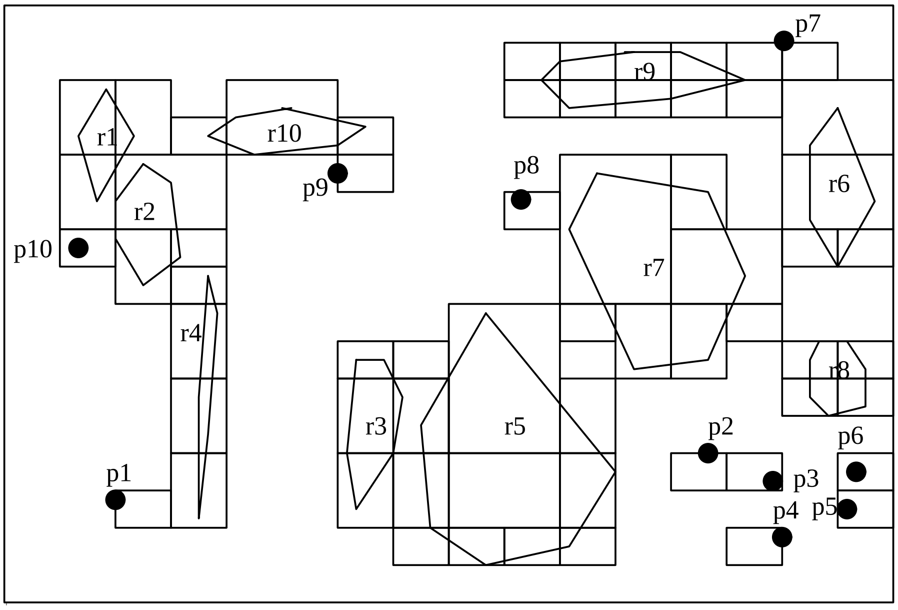


Figure 28: Z-Ordering

5.2 Overlapping Regions

The key idea of the overlapping regions technique is to allow different data buckets in an access method to correspond to *mutually overlapping* subspaces. With this method we can assign any extended object directly and as a whole to one single bucket region. Consider, for instance, the k-d-B-tree for the running example, depicted in Figure 19 (page 29), and one of the polygons given in the scenario (Figure 9, page 15), say r10. r10 overlaps two bucket regions, the one containing p10, c1, and c2, and the other one containing c10 and p9. If we extend one of those regions to accommodate r10, this polygon could be stored in the corresponding bucket. Note, however, that this extension inevitably leads to an overlap of regions.

Search algorithms can be applied almost unchanged. The only differences are due to the fact that the overlap may increase the number of search paths we have to follow. Even a point query may require the investigation of multiple search paths because there may be several subspaces at any index level that include the search point. For range and region queries, the average number of search paths increases as well.

Hence, while functionality is not a problem when using overlapping regions, performance can be. This is particularly relevant when the spatial database contains objects whose size is large relative to the size of the universe. Typical examples are known from geographic applications where one has to represent objects of widely varying size (such as buildings and states) in the same spatial database. Each insertion of a new data object may increase the overlap and therefore the average number of search paths to be traversed per query. Eventually, the overlap between subspaces may become large enough to render the index ineffective because one ends up searching most of the index for each single point query. A well–known example where this degenerate behavior has been observed is the R–tree (Guttman 1984; Greene 1989). Several modifications have been presented to mitigate these problems, including a technique to minimize the overlap (Roussopoulos and Leifker 1985); see Section 5.2.1 for a detailed discussion.

A minor problem with overlapping regions concerns ambiguities during insertion. If we insert a new object, we could in principle enlarge any subspace to accommodate it. To optimize performance, there exist several strategies (Pagel et al. 1993). For example, we could try to find the subspace that causes minimal additional overlap, or the one that requires the least enlargement. If it takes too long to compute the optimal strategy for every insertion, some heuristic may be used.

When a subspace needs to be split, one also tries to find a split that leads to minimal overall overlap. Guttman (1984), Greene (1989), and Beckmann et al. (1990) suggest some heuristics for this problem.

5.2.1 The R-Tree (Guttman 1984)

An *R-tree* corresponds to a hierarchy of nested d-dimensional intervals (boxes). Each node ν of the R-tree corresponds to a disk page and a d-dimensional interval $I^d(\nu)$. If ν is an interior node then the intervals corresponding to the descendants ν_i of ν are contained in $I^d(\nu)$. Intervals at the same tree level may overlap. If ν is a leaf node, $I^d(\nu)$ is the d-dimensional minimum bounding box (MBB) of the objects stored in ν. For each object in turn, ν only stores its MBB and a reference to the complete object description. Other properties of the R-tree include:

- Every node contains between m and M entries unless it is the root. The lower bound m prevents the degeneration of trees and ensures an efficient storage utilization. Whenever the number of a node's descendants drops below m, the node is deleted and its descendants are distributed among the sibling nodes (*tree condensation*). The upper bound M can be derived from the fact that each tree node corresponds to exactly one disk page.
- The root node has at least two entries unless it is a leaf.
- The R-tree is height-balanced, i.e., all leaves are at the same level. The height of an R-tree is at most $\lceil \log_m(N) \rceil$ for N index records ($N > 1$).

Searching in the R-tree is similar to the B-tree. At each index node ν, all index entries are tested whether they intersect the search interval I_s. We then visit all child nodes ν_i with $I^d(\nu_i) \cap I_s \neq \emptyset$. Due to the overlapping region paradigm, there may be several intervals $I^d(\nu_i)$ that satisfy the search predicate. In the worst case, one may have to visit every index page. An example is given in Figure 29 that shows an R-tree for the running example. Remember that the mi denote the MBBs of the polygonal data objects ri. A point query with search point X results in two paths: R8 $\rightarrow$ R4 $\rightarrow$ m7 and R7 $\rightarrow$ R3 $\rightarrow$ m5.

Because the R-tree only manages MBBs, it cannot solve a given search problem completely unless, of course, the actual data objects are interval-shaped. Otherwise the result of an R-tree query is a set of candidate objects, whose actual spatial extent has to be tested for intersection with the search space (cf. Fig. 2, p. 9). This step, which may involve additional disk accesses and considerable computations, has not been taken into account in most published performance analyses (Guttman 1984; Greene 1989).

To insert an object o, we insert the minimum bounding interval $I^d(o)$ and an object reference into the tree. In contrast to searching, we traverse only a single path from the root to the leaf. At each level we choose the child node ν whose corresponding interval $I^d(\nu)$ needs the least enlargement to enclose the data object's interval $I^d(o)$. If several intervals satisfy this criterion, Guttman proposes to select the descendant associated with the smallest interval. As a result, we insert the object only once, i.e., the object is not dispersed over several buckets. Once we have reached the leaf level, we try to insert the object. If this requires an enlargement of the corresponding bucket region, we adjust it appropriately and propagate the change upwards. If there is not enough space left in the leaf, we split it and distribute the entries among the old and the new page. Once again, we adjust each of the new intervals accordingly and propagate the split up the tree.

As for deletion, we first perform an exact match query for the object in question. If we find it in the tree, we delete it. If the deletion causes no underflow we check whether the bounding interval can be reduced in size. If so, we perform this adjustment and propagate it upwards. On the other hand, if the deletion causes node occupation to drop below m, we copy the node content into a temporary node and remove it from the index. We then propagate the node removal up the tree, which typically results in the adjustment of several bounding intervals. Afterwards we reinsert all orphaned entries of the temporary node. Alternatively, we can merge the orphaned entries with sibling entries. In both cases, one may again have to adjust bounding intervals further up the tree.

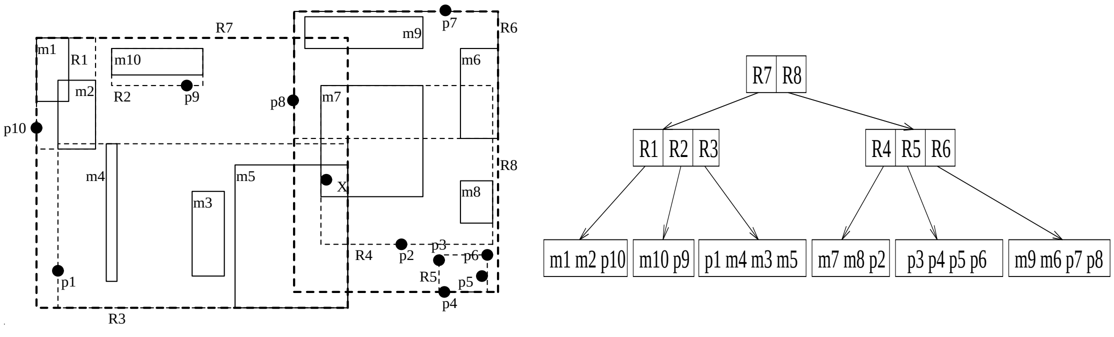


Figure 29: R-Tree

In his original paper, Guttman (1984) discusses various policies to minimize the overlap during insertion. For node splitting, for example, Guttman suggests several algorithms, including a simpler one with linear time complexity and a more elaborate one with quadratic complexity. Later work by other researchers led to the development of more sophisticated policies. The *packed R-tree* (Roussopoulos and Leifker 1985), for example, computes an optimal partitioning of the universe and a corresponding minimal R-tree for a given scenario. However, it requires all data to be known a priori.

Other interesting variants of the R-tree include the *sphere tree* by Oosterom (1990) and the *Hilbert R-tree* by Kamel and Faloutsos (1994). The sphere tree corresponds to a hierarchy of nested d-dimensional spheres rather than intervals. In the Hilbert R-tree, the important feature is that interior nodes also store the largest Hilbert value (cf. Section 4.3) of the data rectangles in the corresponding subtree (in addition to the MBB and a pointer). This enhancement facilitates the insertion of new objects considerably. Together with a revised splitting policy, Kamel and Faloutsos are able to report good performance results for both searches and updates. However, since their splitting policy takes only the Hilbert values of the objects' *centroids* into account, the performance of the Hilbert R-tree is likely to deteriorate in the presence of large extended objects.

Ng and Kameda (1993) discuss how to support concurrency in R-trees by adopting the lock-coupling technique of B-trees (Bayer and Schkolnick 1977) to R-trees. Similarly, Ng and Kameda (1994) and Kronacker and Banks (1995) apply ideas of the B-link tree (Lehman and Yao 1981) to R-trees, yielding a structure dubbed *R-link tree*. Kronacker and Banks empirically demonstrate that the R-link tree is superior to the R-tree using lock-coupling.

5.2.2 The R*-Tree (Beckmann et al. 1990)

Based on a careful study of the R-tree behavior under different data distributions, Beckmann et al. (1990) identified several weaknesses of the original algorithms. In particular, they confirmed the observation of Roussopoulos and Leifker (1985) that the insertion phase is critical for search performance. The design of the *R*-tree* therefore introduces a policy called *forced reinsert*: If a node overflows, they do not split it right away. Rather, they first remove p entries from the node and reinsert them into the tree. The parameter p may vary; Beckmann et al. suggest p to be about 30% of the maximal number of entries per page.

Another issue investigated by Beckmann et al. concerns the node splitting policy. While Guttman's R-tree algorithms only tried to minimize the area that is covered by the bucket regions, the R*-tree algorithms also take the following objectives into account:

- Overlap between bucket regions at the same tree level should be minimized. The less overlap, the smaller the probability that one has to follow multiple search paths.
- Region perimeters should be minimized. The most preferable rectangle is the square, since this is the most compact rectangular representation.
- Storage utilization should be maximized.

The improved splitting algorithm of Beckmann et al. (1990) is based on the plane-sweep paradigm (Preparata and Shamos 1985). In d dimensions, its time complexity is $O(d \cdot n \cdot \log n)$ for a node with n intervals.

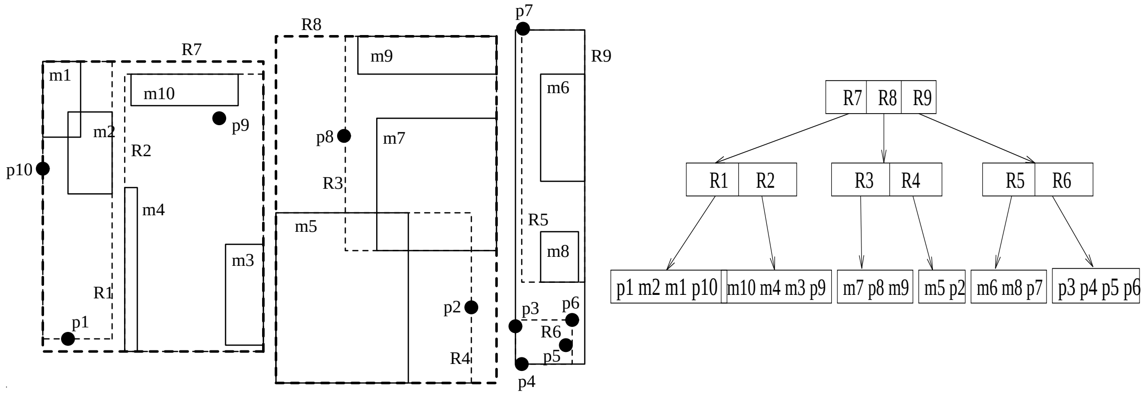


Figure 30: R*-Tree

In summary, the R*-tree differs from the R-tree mainly in the insertion algorithm; deletion and searching are essentially unchanged. Beckmann et al. report performance improvements of up to 50% compared to the basic R-tree. Their implementation also shows that reinsertion may improve storage utilization. In broader comparisons, however, both Hoel and Samet (1992) and Günther and Gaede (1997) found that the CPU time overhead of reinsertion can be substantial, especially for large page sizes; see Section 6 for further details.

One of the major insights of the R*-tree is that node splitting is critical for the overall performance of the access method. Since a naive (exhaustive) approach has time complexity $O(d \cdot 2^n)$ for n given intervals, there is a need for efficient and optimal splitting policies. Becker et al. (1992) proposed a polynomial time algorithm that finds a balanced split, which also optimizes one of several possible objective functions (e.g., minimum sum of areas or minimum sum of perimeters). They assume in their analysis that the intervals are presorted in some specific order.

Recently, Berchtold et al. (1996) proposed a modification of the R-tree, called *X-tree*, that seems to be particularly well suited for indexing high-dimensional data. The X-tree reduces overlap among directory intervals by using a new organization: It postpones node splitting by introducing supernodes, i.e., nodes larger than the usual block size. In order to find a suitable split, the X-tree also maintains the history of previous splits.

5.2.3 The P-Tree (Jagadish 1990c)

In many applications, intervals are not a good approximation of the data objects enclosed. In order to combine the flexibility of polygon-shaped containers with the simplicity of the R-tree, Jagadish (1990c) and Schiwietz (1993) independently proposed different variations of *polyhedral trees* or *P-trees*. To distinguish the two structures, we refer to the P-tree by Jagadish (1990c) as JP-tree and to the P-tree by Schiwietz (1993) as SP-tree.

The *JP-tree* first introduces a variable number m of orientations in the d-dimensional universe, where $m > d$. For instance, in two dimensions ($d = 2$) we may have four orientations ($m = 4$): two parallel to the coordinate axes (i.e., iso-oriented), and two parallel to the two main diagonals. Objects are approximated by minimum bounding polytopes whose faces are parallel to these m orientations. Clearly, the quality of the approximations is positively correlated with m. We can now map the original space into an m-dimensional *orientation space*, such that each (d-dimensional) approximating polytope P^d turns into an m-dimensional interval I^m. Any point inside (outside) P^d maps onto a point inside (outside) I^m, while the opposite is not necessarily true. To maintain the m-dimensional intervals, a large selection of SAMs is available; Jagadish (1990c) suggests the R-tree or R$^+$-tree (cf. Section 5.3.2) for this purpose.

An interesting feature of the JP-tree is the ability to add hyperplanes to the attribute space dynamically without having to reorganize the structure. By projecting the new intervals of the extended orientation space onto the old orientation space, it is still possible to use the old structure. Consequently, we can obtain an R-tree from a higher-dimensional JP-tree structure by dropping all hyperplanes that are not iso-oriented.

The interior nodes of the JP-tree represent a hierarchy of nested polytopes, similar to the R-tree or the cell tree (cf. Section 5.3.3). Polytopes corresponding to different nodes at the same tree level may overlap. For search operations we first compute the minimum bounding polytope of the search region and map it onto an m-dimensional interval. The search efficiency then depends on the chosen PAM. The same applies for deletion.

While the introduction of additional hyperplanes results in a better approximation, it increases the size of the entries, thus reducing the fan-out of the interior nodes. Experiments reported by Jagadish (1990c) suggest that a 10-dimensional orientation space ($m = 10$) is a good choice for storing two-dimensional lines ($d = 2$) with arbitrary orientation. This needs to be compared to a simple MBB approach. Although the latter technique may sometimes render poor approximations, the representation requires only four numbers per line. Storing a 10-dimensional interval, on the other hand, requires 20 numbers, i.e., five times as much. Another drawback of the JP-tree is the fixed orientation of the hyperplanes. Figure 31 shows the running example for $m = 4$.

To overcome the problem of poor filtering, Brodsky et al. (1995) proposed methods for effectively computing a set of optimal axes for separating polyhedra. This work continues the line of work by Jagadish (1990c) in the use of non-standard axes for better filtering.

5.2.4 The P-Tree (Schiwietz 1993)

The P-Tree by Schiwietz, here called *SP-tree*, chooses a slightly different approach for storing polygonal objects. It tries to combine the advantages of the cell tree and the R*-tree for the two-dimensional case, while avoiding the drawbacks of both methods. Basically, the SP-tree is an R-tree whose interior nodes correspond to a nesting of polytopes rather than just rectangles. In general, the number of vertices (and therefore

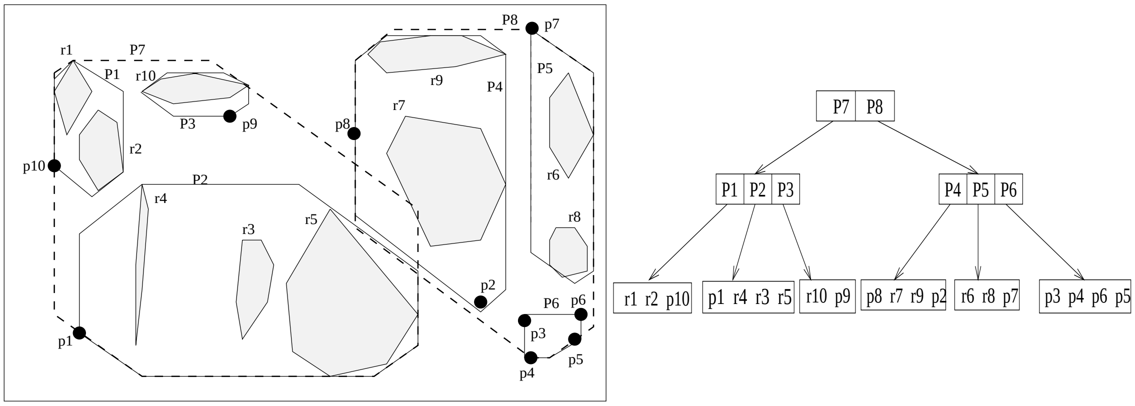


Figure 31: P-Tree (Jagadish 1990c)

the storage requirements) of a polytope is not bounded. Moreover, when used for approximating other objects, the accuracy of the approximation is positively correlated with the number of vertices of the approximating convex polygon. On the other hand, when used as index entries, there should be an upper bound in order to guarantee a minimum fan-out of the interior nodes. To determine a reasonably good compromise between these conflicting objectives, extensive investigations have been conducted by Brinkhoff et al. (1993) and Schiwietz (1993). According to these studies, pentagons or hexagons seem to offer the best tradeoff between storage requirements and approximation quality.

If node splittings or insertions lead to additional vertices, such that some bounding polygons have more vertices than the threshold, the surplus vertices are removed one by one. This leads to a larger area and therefore to a decrease of the quality of the approximation. To reduce overlap between the convex containers, Schiwietz suggests using a method similar to the R*-tree. Furthermore, in order to save storage space and to improve storage utilization, it is possible to restrict the number of orientations for the polygon edges (similar to the JP-tree).

Figure 32 shows the SP-tree for the running example. To our knowledge, no performance results have been reported so far.

5.2.5 The SKD-Tree (Ooi et al. 1987; Ooi 1990)

A variant of the k-d-tree capable of storing spatially extended objects is the *spatial k-d-tree* or *skd-tree*. The skd-tree allows regions to overlap. To keep track of the mutual overlap, we store an upper and a lower bound with each discriminator, representing the maximal extent of the objects in the two subtrees. For example, consider the splitting hyperplane hx1 depicted in Figure 33 and its upper and lower bounds bx1 and bx2, respectively. The solid lines are the splitting hyperplanes and the dashed lines represent the upper and lower bounds of the corresponding subtrees. m3 is the rectangle closest to the hyperplane hx1 without crossing it, thus determining the maximum extension bx1 of the objects in the left (lower) subspace. Similarly, m5 determines the minimum extension bx2 for the right (upper) subspace. If none of the objects placed in the

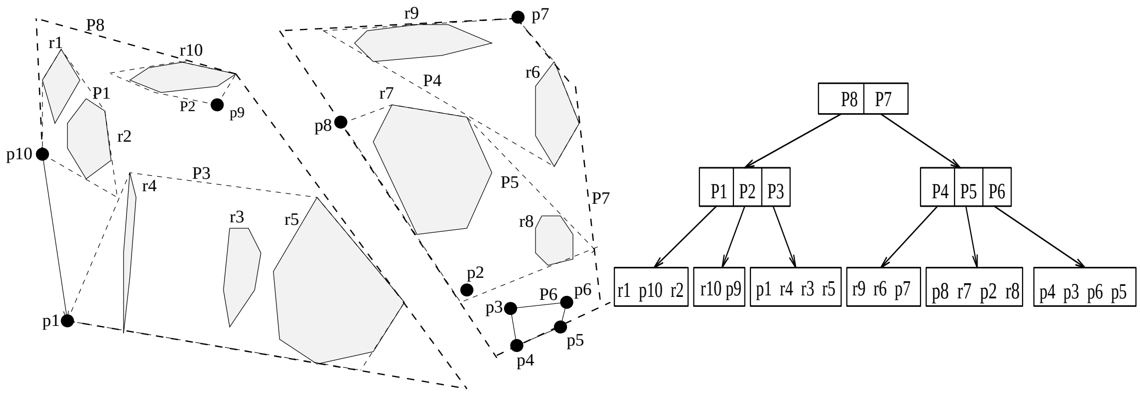


Figure 32: P-Tree (Schiwietz 1993)

corresponding subspace crosses the splitting hyperplane, the lower bound of the upper interval is greater than the discriminator d_k and the upper bound of the lower interval is less than d_k. Leaf nodes of the binary tree contain the minimal bounds (dotted lines) of the objects in the corresponding data page.

Prior to inserting an object o, we determine its centroid and its MBB. By comparing the centroid with the stored discriminators, we determine the child to inspect next. Note that there is no ambiguity. During insertion, we have to adjust the upper and lower bounds for extended objects accordingly. Upon reaching the data node level, we test whether there is enough space available to accommodate the object. If so, we insert the object, otherwise we split the data node and insert the new discriminator into the skd-tree. Likewise, the bounds of the new subspaces need to be adjusted.

As usual, searching starts at the root and corresponds to a top-down tree traversal. At each interior node we check the discriminator and the boundaries to decide which child(ren) to visit next.

Deleting an object starts with an exact match query to determine the correct leaf node. If a deletion causes an underflow, we insert the remaining entries into the sibling data node and remove the splitting hyperplane. If this insertion results in an overflow, we split the page and insert the new hyperplane into the skd-tree. If no merge with a sibling leaf node is possible, we delete that leaf and its parent node. By redirecting the reference of the latter to its sibling (interior) node, we extend the subspace of the sibling. All affected entries are reinserted.

According to the results reported in (Ooi 1990; Ooi et al. 1991) the skd-tree is competitive to the R-tree both in storage utilization and search efficiency.

5.2.6 The GBD-Tree (Ohsawa and Sakauchi 1990)

The GBD-tree (*generalized BD-tree*) is an extension of the BD-tree (Ohsawa and Sakauchi 1983) that allows for secondary storage management and supports the management of extended objects. While the BD-tree is a binary tree, the GBD-tree is a balanced multiway tree that stores spatial objects as a hierarchy of minimum bounding boxes. Each leaf node (bucket) stores the MBBs of those objects whose centroids are

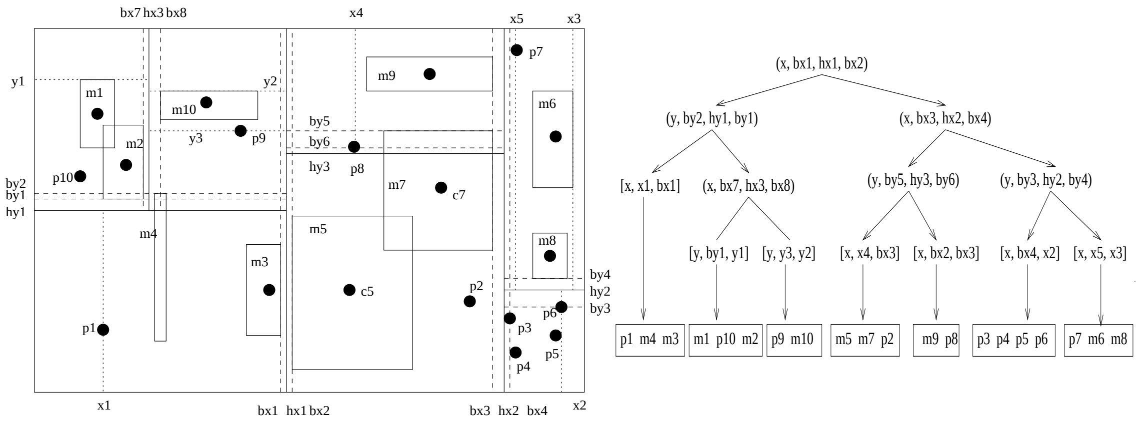


Figure 33: SKD-Tree

contained in the corresponding bucket region. Each interior node stores the MBB of the (usually overlapping) MBBs of its descendants. The intervals are encoded using the same DZ-expressions as described in Section 3.2.3.

The one advantage of the GBD-tree over the R-tree is that insertions and deletions may be processed more efficiently, due to the encoding scheme and the placement by centroid. The latter point enables the GBD-tree to perform an insertion along a single path from the root to a leaf. However, no apparent advantage is gained with respect to search performance. The reported performance experiments (Ohsawa and Sakauchi 1990) compare only storage utilization and *insertion* performance with the R-tree. The most important comparison, that of search performance, is omitted.

Figure 34 depicts a GBD-tree for the running example. The partitioning on the left hand side shows the minimum bounding boxes (dotted or dashed) and the underlying intervals (Peano regions).

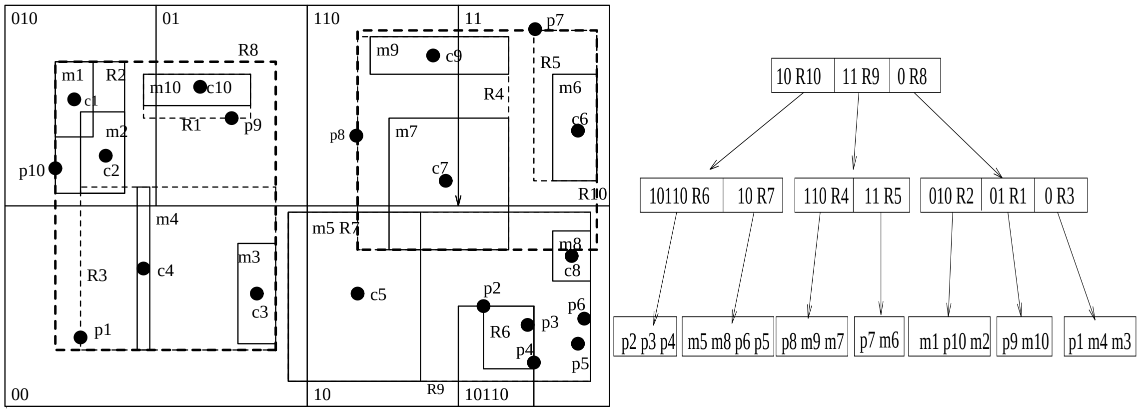


Figure 34: GBD-Tree

Among the approaches similar to the GBD-tree are an extension of the buddy tree by Seeger (1991), and the extension of the BANG file to handle extended spatial objects (Freeston 1989b).

5.2.7 PLOP-Hashing (Kriegel and Seeger 1988; Seeger and Kriegel 1988)

Piecewise linear order preserving (PLOP) hashing (Seeger and Kriegel 1988) is a variant of hashing that allows the storage of extended objects without transforming them into points. An earlier version of this structure (Kriegel and Seeger 1988) was only able to handle multidimensional point data.

PLOP-hashing partitions the universe in a similar way as the grid file; extended objects may span more than one directory cell. Hyperplanes extend along the axes of the data space. For the organization of these hyperplanes, PLOP-hashing uses d binary trees, where d is the dimension of the universe. Each interior node of such a binary tree corresponds to a $(d-1)$-dimensional iso-oriented hyperplane. The leaf nodes represent d-dimensional subspaces forming slices of the universe.

Figure 35 depicts the binary trees for both axes together with the slices formed by them. By using the index entries that are stored in the leaf nodes, we can easily identify the data page we are looking for. To do this efficiently, we have to keep the d binary trees in main memory, similar to the scales of the grid file. For further speed-up, the leaf nodes of each binary tree are linked to each other. In Figure 35 this is suggested by the arrows attached to the leaves of the trees. To handle extended objects, we enlarge the storage representation of each slice by a lower and an upper bound. These bounds indicate the minimum and the maximum extension along the current dimension of all objects stored in the slice at hand.

Insertion is straightforward and similar to the grid file. To avoid ambiguities, PLOP-hashing uses the centroid of the object to determine the data bucket in which to place the object. In the case of node splitting and deletion we have to adjust the respective upper and lower bounds. It should further be noted that PLOP-hashing can easily be modified so that it supports clipping rather than overlapping regions.

Analytical experiments indicate that PLOP-hashing is superior to the R-tree and R^+-tree for uniform data distributions (Seeger and Kriegel 1988).

5.3 Clipping

Clipping–based schemes do not allow any overlaps between bucket regions; they have to be *mutually disjoint.* A typical access method of this kind is the R^+–tree (Stonebraker, Sellis, and Hanson 1986; Sellis, Roussopoulos, and Faloutsos 1987), a variant of the R-tree that allows no overlap between regions corresponding to nodes at the same tree level. As a result, point queries follow a single path starting at the root, which means efficient searches.

The main problems with clipping-based approaches relate to the insertion and deletion of data objects. During insertion, any data object that spans more than one bucket region has to be subdivided along the partitioning hyperplanes. Eventually, several bucket entries have to be created for the same object. Each bucket stores either the geometric description of the complete object *(object duplication)*, or the geometric description of the corresponding fragment with an object reference. In any case, data about the object is dispersed among several data pages *(spanning property)*. The resulting *redundancy* (Orenstein 1989a; Orenstein 1989b; Günther and Gaede 1997) may cause not only an increase in the average search time, but also an increase in the

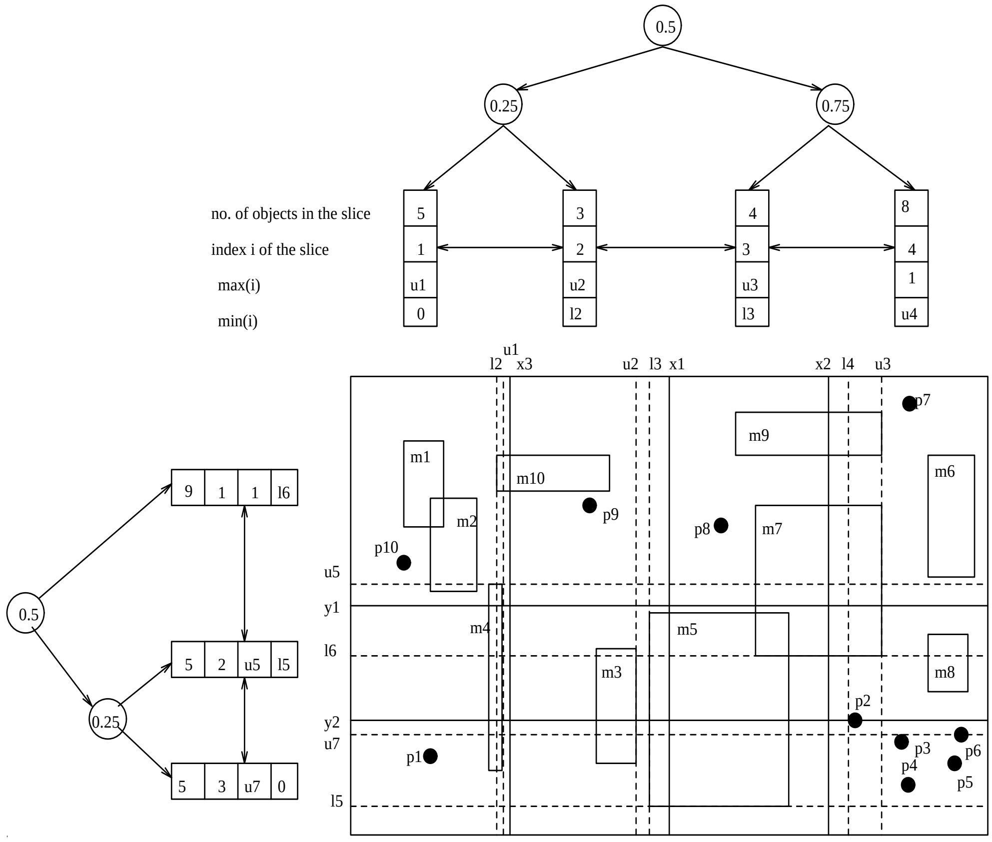


Figure 35: PLOP-Hashing

frequency of bucket overflows.

A second problem applies to clipping-based access methods that do not partition the complete data space. In that case, the insertion of a new data object may lead to the enlargement of several bucket regions. Whenever the object (or a fragment thereof) is passed down to a bucket (or, in the case of a tree structure, an interior node) whose region does not cover it, the region has to be extended. In some cases, such an enlargement is not possible without getting an overlap with other bucket regions; this is sometimes called the *deadlock* problem of clipping. Because overlap is not allowed, we have to redesign the region structure, which can become very complicated. It may in particular cause further bucket overflows and insertions, which can lead to a chain reaction and, in the worst case, a complete breakdown of the structure (Günther and Bilmes 1991). Access methods partitioning the complete data space do not suffer from this problem.

A final problem concerns the splitting of buckets. There may be situations where a bucket (and its corresponding region) has to be split but there is no splitting hyperplane that splits none (or only a few) of the objects in that bucket. The split may then trigger other splits, which may become problematic with increasing size of the database. The more objects are inserted, the higher the probability of splits and the smaller the average size of the bucket regions. New objects are therefore split into a larger number of smaller fragments, which may in the worst case once again lead to a chain reaction. To alleviate these problems, Günther and Noltemeier (1991) suggest storing large objects (which are more likely to be split into a large number of fragments) in special buckets called *oversize shelves*, instead of inserting them into the structure.

5.3.1 The Extended K-D-Tree (Matsuyama, Hao, and Nagao 1984)

One of the earliest extensions of the adaptive k-d-tree that was capable of handling extended objects was the *extended k-d-tree*. In contrast to the skd-tree (Section 5.2.5), the extended k-d-tree is based on clipping. Each interior tree node corresponds to a $(d-1)$-dimensional partitioning hyperplane, represented by the dimension (e.g., x or y) and the splitting coordinate (the *discriminator*). A leaf node corresponds to a rectangular subspace and contains the address of the data page describing that subspace. Data pages may be referenced by multiple leaf nodes.

To insert an object, we start at the root of the k-d-tree. At each interior node, we test for intersection with the stored hyperplane. Depending on the location of the object relative to the hyperplane, we either move on to the corresponding child node, or we clip the object by the hyperplane and follow both branches. This procedure guarantees that we insert the object in all overlapping bucket regions. If a data page cannot accommodate the additional object, we split the page by a new hyperplane. The splitting dimension is perpendicular to the dimension with the greatest extension. After distributing the entries of the data page among the two new pages, we insert the hyperplane into the k-d-tree. Note that this may in turn cause some objects to be split, which may lead to further page overflows. To delete an object, we have to visit all subspaces intersecting the object and delete the stored object identifier. If a data page is empty due to deletion, we remove it and mark all leaf nodes pointing to that page as *NIL*. No merging of sibling nodes is performed.

Figure 36 depicts an extended k-d-tree for the running example. Rectangle m7 has been clipped and inserted into two nodes. Most partitions contain one or two additional bounding hyperplanes (dotted lines) to provide a better localization of the objects in the corresponding subspace.

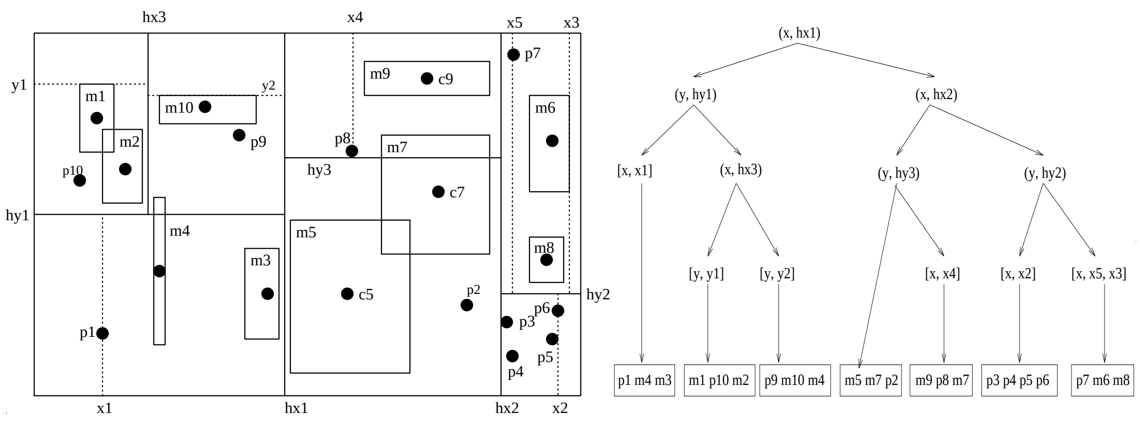


Figure 36: Extended K-D-Tree

5.3.2 The R$^+$-Tree (Stonebraker, Sellis, and Hanson 1986; Sellis, Roussopoulos, and Faloutsos 1987)

To overcome the problems associated with overlapping regions in the R-tree, Sellis et al. introduced an access method called *R$^+$-tree*. Unlike the R-tree, the R$^+$-tree uses clipping, i.e., there is no overlap between index intervals I^d at the same tree level. Objects that intersect more than one index interval have to be stored on several different pages. As a result of this policy, point searches in R$^+$-trees correspond to *single-path* tree traversals from the root to one of the leaves. They therefore tend to be faster than the corresponding R-tree operation. Range searches will usually lead to the traversal of multiple paths in both structures.

When inserting a new object o, we may have to follow multiple paths, depending on the number of intersections of the MBB $I^d(o)$ with index intervals. During the tree traversal, $I^d(o)$ may be split into n disjoint fragments $I_i^d(o)$ ($\bigcup_{i=1}^{n} I_i^d(o) = I^d(o)$). Each fragment is then placed in a different leaf node ν_i. Provided that there is enough space, the insertion is straightforward. If the bounding interval $I^d(o)$ overlaps space that has not yet been covered, we have to enlarge the intervals corresponding to one or more leaf nodes. Each of these enlargements may require a considerable effort because overlaps have to be avoided. In some rare cases, it may not be possible to increase the current intervals in such a way that they cover the new object without some mutual overlap (Günther 1988; Ooi 1990). In case of such a deadlock, some data intervals have to be split and reinserted into the tree.

If a leaf node overflows it has to be split. Node splittings work similarly as in the case of the R-tree. An important difference, however, is that splits may propagate not only up the tree, but also down the tree. The resulting *forced split* of the nodes below may lead to several complications, including further fragmentation of the data intervals; see for example the rectangles m5 and m8 in Figure 37.

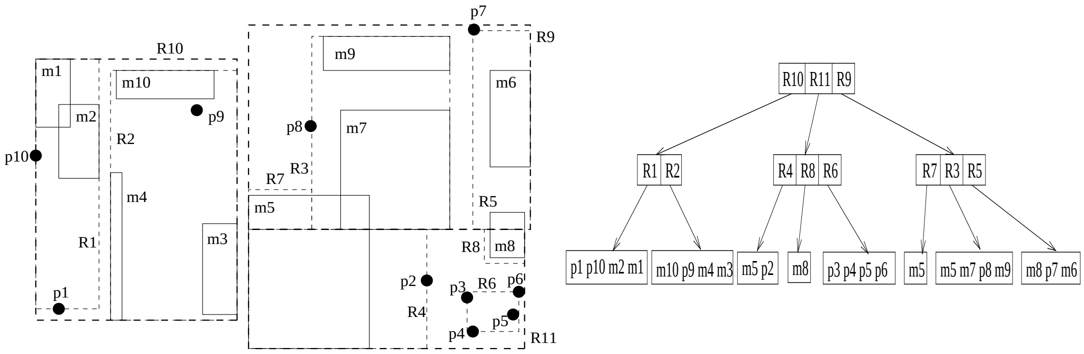


Figure 37: R$^+$-Tree

For deletion, we first locate all the data nodes where fragments of the object are stored and remove them. If storage utilization drops below a given threshold, we try to merge the affected node with its siblings or to reorganize the tree. This is not always possible, which is the reason why the R$^+$-tree cannot guarantee a minimum space utilization.

5.3.3 The Cell Tree (Günther 1988)

The main goal during the design of the *cell tree* (Günther 1988; Günther 1989) was to facilitate searches on data objects of arbitrary shapes, i.e., especially on data objects that are not intervals themselves. The cell tree uses clipping to manage large spatial databases that may contain polygons or higher-dimensional polyhedra. It corresponds to a decomposition of the universe into disjoint convex subspaces. The interior nodes correspond to a hierarchy of nested polytopes and each leaf node corresponds to one of the subspaces (Figure 38). Each tree node is stored on one disk page.

To avoid some of the disadvantages resulting from clipping, the convex polyhedra are restricted to be subspaces of a BSP (Binary Space Partitioning). Therefore we can view the cell tree as a combination of a BSP - and an R^{+}–tree, or as a BSP-tree mapped on paged secondary memory. In order to minimize the number of disk accesses that occur during a search operation, the leaf nodes of a cell tree contain all the information required for answering a given search query; we have to load no pages other than those containing relevant data. This is an important advantage of the cell tree over the R–tree and related structures.

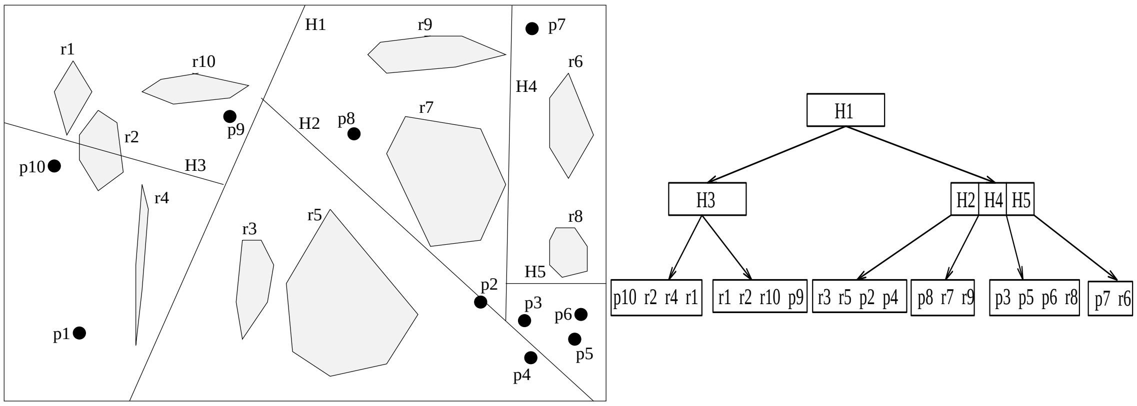


Figure 38: Cell Tree

Before inserting a non-convex object, we decompose it into a number of convex *components* whose union is the original object. The components do not have to be mutually disjoint. All components are assigned the same object identifier and inserted into the cell tree one by one. Due to clipping, we may have to subdivide each component into several *cells* during insertion, because it overlaps more than one subspace. Each cell is stored in one leaf node of the cell tree. If an insertion causes a disk page to overflow, we have to split the corresponding subspace and cell tree node and distribute its descendants among the two resulting nodes. Each split may propagate up the tree.

For point searches, we start at the root of the tree. Using the underlying BSP partitioning, we identify the subspace that includes the search point and continue the search in the corresponding subtree. This step is repeated recursively until we reach a leaf node, where we examine all cells whether they contain the search point. The solution consists of those objects that contain at least one of the cells that qualify. A similar algorithm exists for range searches. A performance evaluation of the cell tree

(Günther and Bilmes 1991) shows that it is competitive with other popular spatial access methods.

Figure 38 shows our running example with five partitioning hyperplanes, each of them stored in the interior nodes. Even though the partitioning by means of the BSP-tree offers more flexibility compared to rectilinear hyperplanes, it may be inevitable to clip objects. In Figure 38, we had to split r2 and insert the resulting cells into two pages.

As all structures based on clipping, the cell tree has to cope with the fragmentation of space, which is becoming increasingly problematic as more objects are inserted into the tree. After some time, most new objects will be split into fragments during insertion. To avoid the negative effects resulting from this fragmentation, Günther and Noltemeier (1991) proposed the concept of *oversize shelves*. Oversize shelves are special disk pages that are attached to the interior nodes of the tree and that accommodate objects which would have been split into too many fragments if they had been inserted regularly. The authors propose a dynamically adjusting threshold for choosing between placing a new object on an oversize shelf or inserting it regularly. Performance results of Günther and Gaede (1997) show substantial improvements compared to the cell tree without oversize shelves.

5.4 Multiple Layers

The multiple layer technique can be regarded as a variant of the overlapping regions approach, because data regions of different layers may overlap. However, there are several important differences: First, the layers are organized in a hierarchy. Second, each layer partitions the complete universe in a different way. Third, data regions within a layer are disjoint, i.e., they do not overlap. Fourth, the data regions do not adapt to the spatial extensions of the corresponding data objects.

In order to get a better understanding of the multi-layer technique, we shall discuss how to insert an extended object. First, we try to find the lowest layer in the hierarchy whose hyperplanes do not split the new object. If there is such a layer, we insert the object into the corresponding data page. If the insertion causes no page to overflow, we are done. Otherwise, we must split the data region by introducing a new hyperplane and distribute the entries accordingly. Objects intersecting the hyperplane have to be moved to a higher layer or an overflow page. As the database becomes populated, the data space of the lower layers becomes more and more fragmented. As a result, large objects keep accumulating on higher layers of the hierarchy or even worse, it is no more possible to insert objects without intersecting existing hyperplanes.

The multi-layer approach seems to offer one advantage compared to the overlapping regions technique: a possibly higher selectivity during searching due to the restricted overlap of the different layers. However, there are also several disadvantages: First, the multi-layer approach suffers from fragmentation, which may render the technique inefficient for some data distributions. Second, certain queries require the inspection of all existing layers. Third, it is not clear how to cluster objects that are spatially close to each other but in different layers. Fourth, there is some ambiguity in which layer to place the object.

Sevcik and Koudas (1996) recently proposed a *static* SAM based on multiple layers, called the *filter tree*. Here, each layer is the result of a regular subdivision of the universe. A new object is assigned to a unique layer, depending on the object's position and extension. Objects within one layer are first sorted by the Hilbert code of their center, then packed into data pages of a given size. Finally, the largest Hilbert code of each data page is inserted into a B-tree.

We continue with a detailed description of two *dynamic* SAMs based on multiple layers.

5.4.1 The Multi-Layer Grid File (Six and Widmayer 1988)

Yet another variant of the grid file capable of handling extended objects is the *multi-layer grid file* (not to be confused with the multilevel grid file of Whang and Krishnamurthy (1985)). The multi-layer grid file consists of an ordered sequence of grid layers. Each of these layers corresponds to a separate grid file with freely positionable splitting hyperplanes that covers the whole universe. A new object is inserted into the first grid file in the sequence that does not imply any clipping of the object. This is an important difference to the twin grid file (see Section 4.1.4), where objects can be moved freely between the two layers. If one of the grid files is extended by adding another splitting hyperplane, those objects that would be split have to be moved to another layer. Figure 39 illustrates a multi-layer grid file with two layers for the running example.

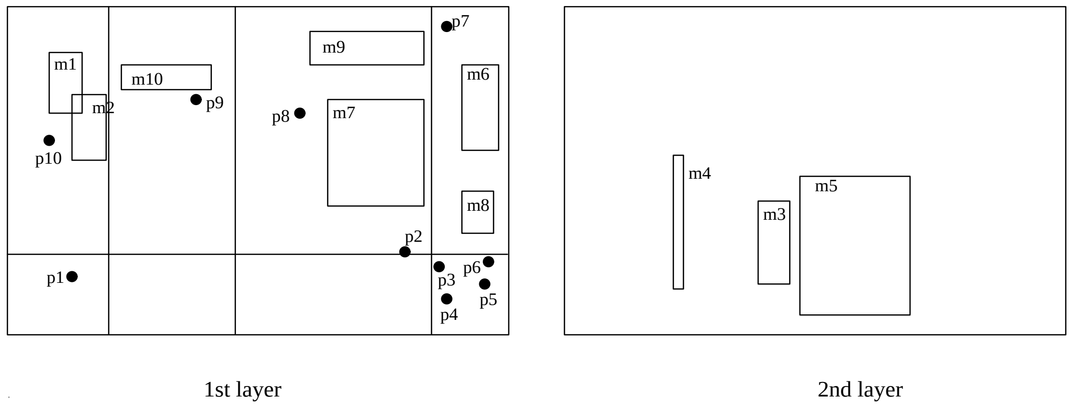


Figure 39: Multi-Layer Grid File

In the multi-layer grid file, the size of the bucket regions typically increases within the sequence, i.e., larger objects are more likely to find their final location in later layers. If a new object cannot be stored in any of the current layers without clipping, a new layer has to be allocated. An alternative is to allow clipping only for the last layer. Six and Widmayer claim that $d+1$ layers are sufficient to store a set of d-dimensional intervals without clipping if the hyperplanes are cleverly chosen.

For an exact match query, we can easily determine from the scales which grid file in the sequence is supposed to hold the search interval. Other search queries, in particular point and range queries, are answered by traversing the sequence of layers and by performing a corresponding search on each grid file. The performance results reported by Six and Widmayer (1988) suggest that the multi-layer grid file is superior to the conventional grid file, using clipping to handle extended objects. Possible disadvantages of the multi-layer grid file include low storage utilization and expensive directory maintenance.

5.4.2 The R-File (Hutflesz et al. 1990)

To overcome some of the problems of the multi-layer grid file, Hutflesz et al. (1990) proposed an alternative structure for managing sets of rectangles, called the *R-file*; see Figure 40 for an example. In order to avoid the low storage utilization of the multi-layer grid file, the R-file uses a single directory. The universe is partitioned similarly to the BANG file: Splitting hyperplanes cut the universe recursively into equal parts, and z-ordering is used to encode the resulting bucket regions. In contrast to the BANG file, however, there are no excisions. Bucket regions may overlap, and there is no clipping. Each data interval is stored in the bucket with the smallest region that contains it entirely; overflow pages may be necessary in some cases.

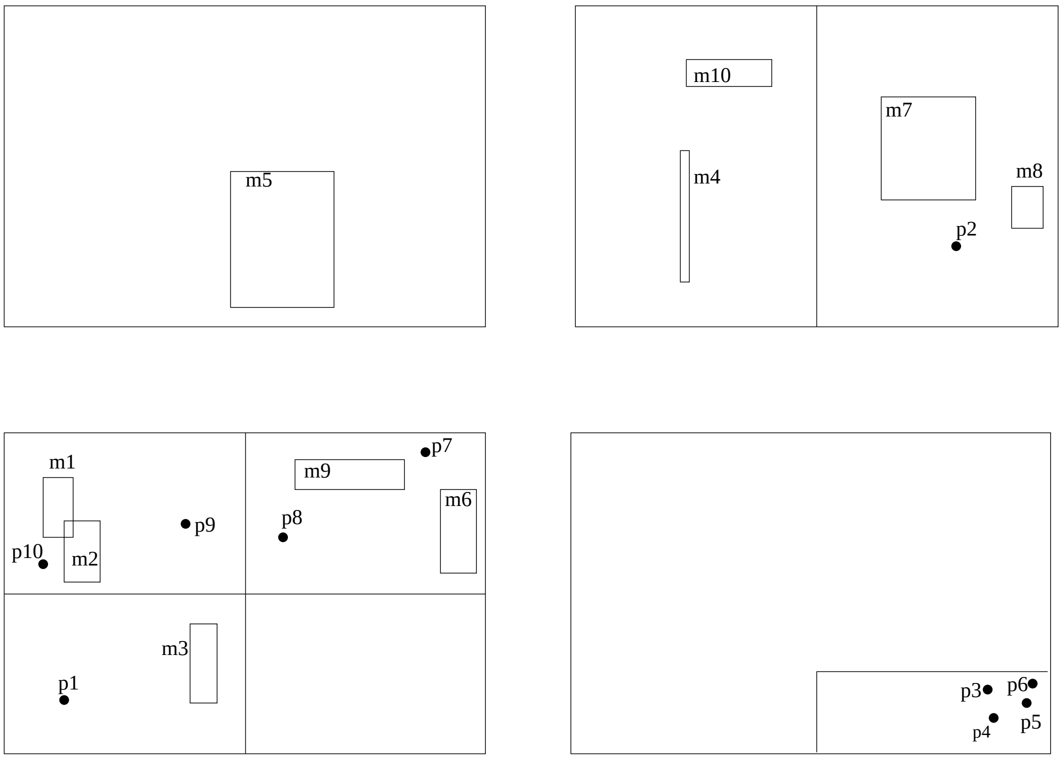


Figure 40: R-File

An interesting feature of the R-file is its splitting algorithm. Rather than cutting a bucket region into two halves, we retain the original bucket region and create a new bucket for *one* of the two halves of that original region. Data intervals are then assigned to the new bucket if and only if they are completely contained in the corresponding region. The half is chosen in such a way that the distribution of data intervals between the two resulting buckets is most even. Once a region has been split, it may subsequently be split again, using the same algorithm. Since objects that are located near the middle of the universe are likely to intersect the partitioning hyperplanes, they are often assigned to the cell region corresponding to the whole universe. Thus objects in that cell tend to cluster near the splitting hyperplanes (cf. rectangle r5 in Figure 40).

To avoid searching dead space, the R-file maintains minimum enclosing boxes of the stored objects, called *search regions*. As shown by Hutflesz et al. (1990), this feature, together with the z-encoding of the partitions, make the R-file competitive to the R-tree. One drawback of the R-file is the fact that it partitions the entire space, whereas the R-tree only indexes the part of the universe that contains objects. For data distributions that are non-uniform, the R-file will therefore often perform poorly. This disadvantage is something that the R-file shares with the grid file. Widmayer (1991) also notes that the R-file is "algorithmically complicated."

6 Comparative Studies

In this section, we give a brief overview of theoretical and experimental results on the comparison of different access methods. Unfortunately, the number of such evaluations, especially theoretical analyses, is rather limited.

Greene (1989) compares the search performance of the R-tree, the k-d-B-tree, and the R^+-tree for 10,000 uniformly distributed rectangles of varying size. Query parameters include the size of the query rectangles and the page size. Greene's study shows that the k-d-B-tree can never really compete with the two R-tree variants. On the other hand, there is not much difference between the R^+-tree and the R-tree, even though the former is significantly more difficult to code. As expected, the R^+-tree performs better when there is less overlap between the data rectangles.

Kriegel et al. (1990) present an extensive experimental study of access method performance for a variety of point distributions. The study involves four point access methods: the hB-tree, the BANG file, the two-level grid file, and the buddy tree. The authors decided not to include PLOP-hashing since its performance suffers considerably for non-uniform data. The zkdB^+-tree by Orenstein and Merrett (1984) was also not included since the authors considered both the BANG file and the hB-tree as improvements of that strategy. Finally, Kriegel et al. did not include quantile hashing although they claim in (Kriegel and Seeger 1987; Kriegel and Seeger 1989) that this structure is very efficient for non-uniform data.

According to the benchmarks, the buddy tree and, to some degree, the BANG file outperform all other structures. The reported results show in an impressive way how the performance of the studied access methods varies with different data distributions and query range sizes. For clustered data and a query range of size 10 % of the size of the universe, there is almost no performance difference between the buddy tree and the BANG file. If the size of the query range drops to only 0.1 % of the size of the universe, the buddy tree performs about twice as fast.

For extended objects, Kriegel et al. compared the R-tree and PLOP-hashing with the buddy tree and the BANG file. The latter two techniques were enhanced by the transformation technique to handle rectangles. Once again, the buddy tree and the

BANG file outperformed the other two access methods for nearly all data distributions. Note that the benchmarks measured only the number of page accesses but not the CPU time.

Beckmann et al. (1990) compared the R*-tree with several variants of the R-tree for a variety of data distributions. Besides the performance of the different structures for point, intersection and enclosure queries for varying query region sizes, they also compared spatial join performance. The R*-tree is the clear winner for all data distributions and queries, and it also has the best storage utilization and insertion times. A comparison for point data confirms these results. Similar to previous performance measurements, only the number of disk accesses is measured. A related study by Kamel and Faloutsos (1994) finds even better search results for the Hilbert R-tree, while updates take about the same time as for the R*-tree. The impact of global clustering on the search performance of the R*-tree was investigated by Brinkhoff and Kriegel (1994). Kamel, Khalil, and Kouramajian (1996) use Hilbert codes for bulk insertion into dynamic R*-trees.

Seeger (1991) studied the relative performance of clipping, overlapping regions, and transformation techniques, implemented on top of the buddy tree. He also included the two-level grid file and the R*-tree in the comparison. The buddy tree with clipping and the grid file failed completely for certain distributions, since they produced unmanageably large files. The transformation technique supports fast insertions at the expense of low storage utilization. The R*-tree, on the other hand, requires fairly long insertion times, but offers good storage utilization. For intersection and containment queries, the buddy tree combined with overlapping regions is continuously superior to the buddy tree with transformation. The performance advantage of the overlapping regions technique decreases for larger query regions, even though the buddy tree with transformation never outperforms the buddy tree with overlapping regions. When the data set contains uniformly distributed rectangles of varying size, the buddy tree with clipping outperforms the other techniques for intersection and enclosure queries. For some queries the buddy tree with overlapping performs slightly better than the R*-tree.

Ooi (1990) compares a static and a dynamic variant of the skd-tree with the packed R-tree described by Roussopoulos and Leifker (1985). For large page sizes, the skd-tree clearly outperforms the R-tree in terms of page accesses per search operation. The space requirements of the skd-tree, however, are higher than those of the R-tree. Since the skd-tree stores the extended objects by their centroid, containment queries are answered more efficiently than by the R-tree. This behavior is clearly reflected in the performance results. A comparison with the extended k-d-tree, enhanced by overflow pages, suggests that the skd-tree is superior, although the extended k-d-tree (which is based on clipping) performs rather well for uniformly distributed data.

Günther and Bilmes (1991) compare the R-tree to two clipping-based access methods, the cell tree and the R$^+$-tree. Different from most studies, the data sets consist of convex polygons instead of just rectangles. The cell tree requires up to two times more space than its competitors. On the other hand, the average number of page accesses per search operation is less than for the other two access methods. Moreover, this advantage tends to increase with the size of the database and the size of the query regions. Besides measurements on the number of page faults, CPU time measurements are also given.

Günther and Gaede (1997) compare the original cell tree as presented in (Günther 1989) with the cell tree with oversize shelves (Günther and Noltemeier 1991), the R*-tree (Beckmann et al. 1990) and the hB-tree (Lomet and Salzberg 1989) for some real cartographic data. There is a slight performance advantage of the cell tree with oversize shelves compared to the R*-tree and the hB-tree, but a major difference to the original cell tree. An earlier comparison using artificially generated data can be found in (Günther 1991). Both studies suggest that oversize shelves may lead to significant improvements for access methods with clipping.

Oosterom (1990) compares the query times of his KD2B-tree and the sphere tree with the R-tree for different queries. The KD2B-tree is a paged version of the KD2-tree, which in turn is a variant of the k-d-tree. The two structures differ in two aspects: First, each interior node stores two iso-oriented lines to allow for overlap and gaps. Second, the corresponding partition lines do not clip, i.e., an object is handled as a unit. The KD2B-tree outperforms the R-tree for all queries, whereas the sphere tree is inferior to the R-tree.

Hoel and Samet (1992) compare the performance of the PMR-quadtree (Nelson and Samet 1987), the R*-tree, and the R^+-tree for indexing line segments. The R^+-tree shows the best insertion performance, whereas the R*-tree occupies the least space. However, the insertion behavior of the R^+-tree heavily depends on the page size as opposed to the PMR-quadtree. The performance of all compared structures is about the same, even though the PMR-quadtree shows some slight performance benefits. Although the R*-tree is more compact than the other structures, its search performance is not as good as that of the R^+-tree for line segments. Unfortunately, Hoel and Samet do not report the overall performance times for the different queries.

Peloux, Reynal, and Scholl (1994) carried out a similar performance comparison of two quadtree variants, a variant of the R^+-tree, and the R*-tree. What makes their study different is that all structures have been implemented on top of a commercial object-oriented system using the application programmer interface. A further difference to Hoel and Samet (1992) is that Peloux et al. used polygons rather than line segments as test data. Furthermore, they report the various times for index traversal, loading polygons, etc. Besides showing that the R^+-tree and a quadtree variant based on Hierarchical EXCELL (Tamminen 1983) outperform the R*-tree for point queries, they clearly demonstrate that the database system must provide some means for physical clustering. Otherwise, reading a single index page may induce several page faults.

Smith and Gao (1990) compare the performance of a variant of the zkdB^+-tree, the grid file, the R-tree, and the R^+-tree for insertions, deletions and search operations. They also measured storage utilization. The conclusion of their experiments is that z-ordering and the grid file perform well for insertions and deletions, but deliver a poor search performance. R- and R^+-trees, in contrast, offer moderate insertion and deletion performance but superior search performance. Although the R^+-tree performs slightly better than the R-tree for search operations, the authors conclude that the R^+-tree is not a good choice for general purpose applications, due to its potentially poor space utilizaton.

Hutflesz et al. (1990) showed that the R-file has a 10-20 % performance advantage over the R-tree on a data set containing 48,000 rectangles with a high degree of overlap (each point in the database was covered by 5.78 rectangles on the average).

Further experimental studies on the R-tree and related structures can be found in (Frank and Barrera 1989; Kamel and Faloutsos 1992; Kolovson and Stonebraker 1991).

Since splitting of data buckets is an important operation in many structures, Henrich and Six (1991) studied several split strategies. Their theoretical analysis is verified by means of the LSD-tree. They also provide some performance results for the R-tree, which uses their splitting strategy in comparison to the otherwise unchanged R-tree. An empirical performance comparison of the R-tree with an improved variant of z-hashing, called layered z-hashing or *lz-hashing* (Hutflesz et al. 1988a), can be found in (Hutflesz, Widmayer, and Zimmermann 1991). The proposed structure needs significantly less seek operations than the R-tree; average storage utilization is higher.

Jagadish (1990a) studies the properties of different space-filling curves (z-ordering, Gray-coding, and Hilbert-curve). By means of theoretical considerations as well as by experimental tests, he concludes that the Hilbert mapping from multidimensional space to a line is superior to other space-filling curves. These results are in accordance with those of Abel and Mark (1990)

When trying to summarize all those experimental comparisons, the following multidimensional access methods seem to be among the best performing ones (in alphabetical order):

- buddy (hash) tree (Seeger and Kriegel 1990)
- cell tree with oversize shelves (Günther and Gaede 1997)
- Hilbert R-tree (Kamel and Faloutsos 1994)
- KD2B-tree (Oosterom 1990)
- PMR-quadtree (Nelson and Samet 1987)
- R^+-tree (Sellis, Roussopoulos, and Faloutsos 1987)
- R^*-tree (Beckmann, Kriegel, Schneider, and Seeger 1990)

It cannot be emphasized enough, however, that any such ranking needs to be used with great care. Clever programming can often make up for inherent deficiencies of an access method. Other factors of unpredictable impact include the hardware used, the settings of the operating system, and the data sets. Note also that our list does not take into account access methods for which no comparative analyses have been published.

As the preceding discussion shows, although numerous experimental studies exist, they are hardly comparable. Theoretical studies may bring some more objectivity to this discussion. The problem with such studies is that they are usually very hard to perform if one wants to stick to realistic modeling assumptions. For that reason, there are only few theoretical results on the comparison of multidimensional access methods.

Regnier (1985) and Becker (1992) investigated the grid file and some of its variants. The most complete theoretical analysis of range trees can be found in (Overmars et al. 1990; Smid and Overmars 1990). Günther and Gaede (1997) present a theoretical analysis of the cell tree. Recent analyses show that the theory of fractals seems to be particularly suitable for modeling the behavior of SAMs if the data distribution is non-uniform (Faloutsos and Kamel 1994; Belussi and Faloutsos 1995; Faloutsos and Gaede 1996; Papadopoulos and Manolopoulos 1997).

Some more analytical work exists on the R-tree and related methods. A comparison of the R-tree and the R$^+$-tree has been published by Faloutsos et al. (1987). Recently, Pagel et al. (1993) presented an interesting probabilistic model of window query performance for the comparison of different access methods independent of implementation details. Among other things, their model reveals the importance of the perimeter as a criterion for node splitting, which has been intuitively anticipated by the inventors of the R*-tree (Beckmann, Kriegel, Schneider, and Seeger 1990). The central formula of Pagel et al. (1993) to compute the number of disk accesses in an R-tree has been found independently by Kamel and Faloutsos (1993). Faloutsos and Kamel (1994) later refined this formula by using properties of the data set. More recently, Theodoridis and Sellis (1996) proposed a theoretical model to determine the number of disk accesses in an R-tree that only requires two parameters: the amount of data and the density in the data space. Their model also extends to non-uniform distributions.

In pursuit of an implementation-independent comparison criterion for access methods, Pagel et al. (1995) suggest to use the degree of clustering. As a lower bound they assume the optimal clustering of the static situation, i.e., if the complete data set has been exposed beforehand. Incidentally, the significance of clustering for access methods has been demonstrated in numerous empirical investigations as well (Jagadish 1990a; Kamel and Faloutsos 1993; Brinkhoff and Kriegel 1994; Kumar 1994b; Ng and Han 1994).

In the area of constraint database systems (see (Gaede and Wallace 1997) for a recent survey) a number of interesting papers related to multidimensional access methods have been published. Kanellakis et al. (1993), for example, presented a semi-dynamic structure which guarantees certain worst-case bounds for space, search and insertion. Subramanian and Ramaswamy (1995) and Hellerstein et al. (1997) complement this work by proving some important lower and upper bounds. Sexton (1997) and Stuckey (1997) look at indexing from a language point of view. Their work can be regarded as a generalization of work by Hellerstein et al. (1995), who proposed a generic framework for modeling hierarchical access methods.

7 Conclusions

Research in spatial database systems has resulted in a multitude of spatial access methods. Even for experts it becomes more and more difficult to recognize their merits and faults, since every new method seems to claim superiority to at least one access method that has been published previously. This survey did not try to resolve this problem but rather to give an overview of the pros and cons of a variety of structures. It will come as no surprise to the reader that at present no access method has proven itself to be superior to all its competitors in whatever sense. Even if one benchmark declares one structure as the clear winner, another benchmark may prove the same structure as inferior.

But why are such comparisons so difficult? Because there are so many different criteria to define optimality, and so many parameters that determine performance. Both time and space efficiency of an access method strongly depend on the data to be processed and the queries to be asked. An access method that performs reasonably well for iso-oriented rectangles may fail for arbitrarily oriented lines. Strongly correlated data may render an otherwise fast access method irrelevant for any practical application. An index that has been optimized for point queries may be highly inefficient for arbitrary region queries. Large numbers of insertions and deletions may deteriorate a structure that is efficient in a more static environment.

The initiative of Kriegel et al. (1990) to set up a standardized testbed for benchmarking and comparing access methods under different conditions is an important step into the right direction. But note that clever programming can often make up for inherent deficiencies of a structure (and vice versa). Other factors of unpredictable impact are the programming language used, the hardware, buffer size, page size, data set, etc. Hence, it is far from easy to compare or rank different access methods. Experimental benchmarks need to be studied with care and can only be a first indicator for usability.

When it comes to technology transfer, i.e. to the use of access methods in commercial products, most vendors resort to structures that are easy to understand and implement. Z-ordering (Oracle Inc. 1995) and R-trees (Informix Inc. 1997) are typical examples. Performance seems to be of minor importance for the selection, which comes as no surprise given the relatively small differences among methods in virtually all published analyses. The tendency is rather to take a structure that is simple and robust, and to optimize its performance by a highly tuned implementation and tight integration with other system components.

Nevertheless, the implementation and experimental evaluation of access methods is essential as it often reveals deficiencies and problems that are not obvious from the design or a theoretical model. In order to make such comparative evaluations both easier to perform and easier to verify, it is essential to provide platform-independent access to the implementations of a broad variety of access methods. Some extensions of the World Wide Web, including our own MMM project (Günther et al. 1997), may provide the right technological base for such a paradigm change. Once every published paper includes a URL (Uniform Resource Locator), i.e., an Internet address that points to an implementation, possibly with a standardized user interface, transparency will increase substantially. Until then, most users will have to rely on general wisdom and their own experiments to select an access method that provides the best fit for their current application.

Acknowledgements

While working on this survey, we had the pleasure to discuss with many colleagues about their work. Special thanks go to D. Abel, A. Buchmann, C. Faloutsos, A. Frank, M. Freeston, J. C. Freytag, J. Hellerstein, C. Kolovson, H.-P. Kriegel, J. Nievergelt, J. Orenstein, P. Picouet, W.-F. Riekert, D. Rotem, J.-M. Saglio, B. Salzberg, H. Samet, M. Schiwietz, R. Schneider, M. Scholl, B. Seeger, T. Sellis, A. P. Sexton, and P. Widmayer. We would also like to thank the referees for their detailed and insightful comments.

References

Abel, D. J. and D. M. Mark (1990). A comparative analysis of some two-dimensional orderings. *Int. J. Geographical Information Systems 4*(1), 21–31.

Abel, D. J. and J. L. Smith (1983). A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision 24*, 1–13.

Aref, W. G. and H. Samet (1994). The spatial filter revisited. In *Proc. 6th Int. Symp. on Spatial Data Handling*, pp. 190–208.

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

Bayer, R. and M. Schkolnick (1977). Concurrency of operations on B-trees. *Acta Informatica 9*, 1–21.

Becker, B., P. Franciosa, S. Gschwind, T. Ohler, F. Thiem, and P. Widmayer (1992). Enclosing many boxes by an optimal pair of boxes. In A. Finkel and M. Jantzen (Eds.), *Proc. STACS'92* , Number 525 in LNCS, Berlin/Heidelberg/New York, pp. 475–486. Springer–Verlag.

Becker, L. (1992). *A New Algorithm and a Cost Model for Join Processing with the Grid File*. Ph. D. thesis, Universität-Gesamthochschule Siegen, Germany.

Beckmann, N., H.-P. Kriegel, R. Schneider, and B. Seeger (1990). The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 322–331.

Belussi, A. and C. Faloutsos (1995). Estimating the selectivity of spatial queries using the 'correlation' fractal dimension. In *Proc. 21st Int. Conf. on Very Large Data Bases*, pp. 299–310.

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

Bentley, J. L. (1979). Multidimensional binary search in database applications. *IEEE Trans. Software Eng. 4*(5), 333–340.

Bentley, J. L. and J. H. Friedman (1979). Data structures for range searching. *ACM Computing Surveys 11*(4), 397–409.

Berchtold, S., D. Keim, and H.-P. Kriegel (1996). The X-tree: An index structure for high-dimensional data. In *Proc. 22nd Int. Conf. on Very Large Data Bases*, Bombay, India, pp. 28–39.

Blanken, H., A. Ijbema, P. Meek, and B. van den Akker (1990). The generalized grid file: Description and performance aspects. In *Proc. 6th IEEE Int. Conf. on Data Eng.*, pp. 380–388.

Brinkhoff, T. (1994). *Der Spatial Join in Geo-Datenbanksystemen*. Ph. D. thesis, Ludwig-Maximilians-Universität München, Germany. In German.

Brinkhoff, T. and H.-P. Kriegel (1994). The impact of global clustering on spatial database systems. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pp. 168–179.

Brinkhoff, T., H.-P. Kriegel, and R. Schneider (1993). Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *Proc. 9th IEEE Int. Conf. on Data Eng.*, pp. 40–49.

Brinkhoff, T., H.-P. Kriegel, R. Schneider, and B. Seeger (1994). Multi-step processing of spatial joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 197–208.

Brinkhoff, T., H.-P. Kriegel, and B. Seeger (1993). Efficient processing of spatial joins using R-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 237–246.

Brodsky, A., C. Lassez, J.-L. Lassez, and M. J. Maher (1995). Seperability of polyhedra for optimal filtering of spatial and constraint data. In *Proc. 14th ACM SIGACT–SIGMOD–SIGART Symp. on Principles of Database Systems*, San Jose, CA, pp. 54 – 64.

Burkhard, W. (1984). Index maintenance for non-uniform record distributions. In *Proc. 3rd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pp. 173–180.

Burkhard, W. A. (1983). Interpolation–based index maintainance. *BIT 23*, 274–294.

Chen, L., R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani (1995). Access to multidimensional datasets on tertiary storage systems. *Information Systems 20*(2), 155–183.

Comer, D. (1979). The ubiquitous B–tree. *ACM Computing Surveys 11*(2), 121–138.

Dandamudi, S. P. and P. G. Sorenson (1986). Algorithms for BD-trees. *Software – Practice and Experience 16*(2), 1077–1096.

Dandamudi, S. P. and P. G. Sorenson (1991). Improved partial-match search algorithms for BD-trees. *The Computer Journal 34*(5), 415–422.

Egenhofer, M. (1989). *Spatial query languages*. Ph. D. thesis, University of Maine, Orono. Ph.D. Thesis.

Egenhofer, M. (1994). Spatial SQL: A query and presentation language. *IEEE Trans. Knowledge and Data Eng. 6*(1), 86–95.

Evangelidis, G. (1994). *The hB^{Π}-Tree: A Concurrent and Recoverable Multi-Attribute Index Structure*. Ph. D. thesis, Northeastern University, Boston, MA.

Evangelidis, G., D. Lomet, and B. Salzberg (1995). The hB^{Π}-tree: A modified hB-tree supporting concurrency, recovery and node consolidation. In *Proc. 21st Int. Conf. on Very Large Data Bases*, pp. 551–561.

Fagin, R., J. Nievergelt, N. Pippenger, and R. Strong (1979). Extendible hashing: A fast access method for dynamic files. *ACM Trans. Database Systems 4*(3), 315–344.

Faloutsos, C. (1986). Multiattribute hashing using Gray-codes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 227–238.

Faloutsos, C. (1988). Gray-codes for partial match and range queries. *IEEE Trans. Software Eng. 14*, 1381–1393.

Faloutsos, C. and V. Gaede (1996). Analysis of n-dimensional quadtrees using the Hausdorff fractal dimension. In *Proc. 22nd Int. Conf. on Very Large Data Bases*, Bombay, India, pp. 40–50.

Faloutsos, C. and I. Kamel (1994). Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proc. 13th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pp. 4–13.

Faloutsos, C. and Y. Rong (1991). DOT: A spatial access method using fractals. In *Proc. 7th IEEE Int. Conf. on Data Eng.*, pp. 152–159.

Faloutsos, C. and S. Roseman (1989). Fractals for secondary key retrieval. In *Proc. 8th ACM SIGACT-SIGMOD–SIGART Symp. on Principles of Database Systems*, pp. 247–252.

Faloutsos, C., T. Sellis, and N. Roussopoulos (1987). Analysis of object oriented spatial access methods. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 426–439.

Finkel, R. and J. L. Bentley (1974). Quad trees: A data structure for retrieval of composite keys. *Acta Informatica 4*(1), 1–9.

Flajolet, P. (1983). On the performance evaluation of extendible hashing and trie searching. *Acta Informatica 20*, 345–369.

Frank, A. and R. Barrera (1989). The fieldtree: A data structure for geographic information systems. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang (Eds.), *Design and Implementation of Large Spatial Database Systems*, Number 409 in LNCS, Berlin/Heidelberg/New York, pp. 29–44. Springer–Verlag.

Freeston, M. (1987). The BANG file: A new kind of grid file. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 260–269.

Freeston, M. (1989a). Advances in the design of the BANG file. In *Proc. 3rd Int. Conf. on Foundations of Data Organization and Algorithms*, Number 367 in LNCS, Berlin/Heidelberg/New York, pp. 322–338. Springer–Verlag.

Freeston, M. (1989b). A well–behaved structure for the storage of geometric objects. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang (Eds.), *Design and Implementation of Large Spatial Database Systems*, Number 409 in LNCS, Berlin/Heidelberg/New York, pp. 287–300. Springer–Verlag.

Freeston, M. (1995). A general solution of the n-dimensional B-tree problem. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 80–91.

Freeston, M. (1997). On the complexity of BV-tree updates. In V. Gaede, A. Brodsky, O. Günther, V. Vianu, and M. Wallace (Eds.), *Proc. CDB'97 and CP'96 Workshop on Constraint Databases and their Application*, Number 1191 in LNCS, Berlin/Heidelberg/New York, pp. 282–293. Springer–Verlag.

Fuchs, H., G. D. Abram, and E. D. Grant (1983). Near real–time shaded display of rigid objects. *Computer Graphics 17*(3), 65–72.

Fuchs, H., Z. Kedem, and B. Naylor (1980). On visible surface generation by a priori tree structures. *Computer Graphics 14*(3).

Gaede, V. (1995a). Geometric information makes spatial query processing more efficient. In *Proc. 3rd ACM Int. Workshop on Advances in Geographic Information Systems (ACM-GIS'95)*, Baltimore, Maryland, USA, pp. 45–52.

Gaede, V. (1995b). Optimal redundancy in spatial database systems. In M. J. Egenhofer and J. R. Herring (Eds.), *Proc. 4th Int. Symp. on Spatial Databases (SSD'95)*, Number 951 in LNCS, Berlin/Heidelberg/New York, pp. 96–116. Springer-Verlag.

Gaede, V., A. Brodsky, O. Günther, V. Vianu, and M. Wallace (Eds.) (1997). *Proc. CDB'97 and CP'96 Workshop on Constraint Databases and their Application.* Number 1191 in LNCS. Berlin/Heidelberg/New York: Springer–Verlag. Selected papers.

Gaede, V. and W.-F. Riekert (1994). Spatial access methods and query processing in the object-oriented GIS GODOT. In *Proc. of the AGDM'94 Workshop*, Delft, The Netherlands, pp. 40–52. Netherlands Geodetic Commission.

Gaede, V. and M. Wallace (1997). An informal introduction to constraint databases. In *Proc. CDB'97 and CP'96 Workshop on Constraint Databases and their Application*, Number 1191 in LNCS, Berlin/Heidelberg/New York, pp. 7–52. Springer–Verlag.

Garg, A. K. and C. C. Gotlieb (1986). Order-preserving key transformation. *ACM Trans. Database Systems 11*(2), 213–234.

Greene, D. (1989). An implementation and performance analysis of spatial data access methods. In *Proc. 5th IEEE Int. Conf. on Data Eng.*, pp. 606–615.

Günther, O. (1988). *Efficient Structures for Geometric Data Management.* Number 337 in LNCS. Berlin/Heidelberg/New York: Springer–Verlag.

Günther, O. (1989). The cell tree: An object-oriented index structure for geometric databases. In *Proc. 5th IEEE Int. Conf. on Data Eng.*, pp. 598–605.

Günther, O. (1991). Evaluation of spatial access methods with oversize shelves. In G. Gambosi, M. Scholl, and H.-W. Six (Eds.), *Geographic Database Management Systems*, pp. 177–193. Berlin/Heidelberg/New York: Springer-Verlag.

Günther, O. (1993). Efficient computation of spatial joins. In *Proc. 9th IEEE Int. Conf. on Data Eng.*, pp. 50–59.

Günther, 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.

Günther, O. and A. Buchmann (1990). Research issues in spatial databases. *ACM SIGMOD Record 19*(4), 61–68.

Günther, O. and V. Gaede (1997). Oversize shelves: A storage management technique for large spatial data objects. *Int. J. Geographical Information Systems 11*(1), 5–32.

Günther, O., R. Müller, P. Schmidt, H. Bhargava, and R. Krishnan (1997). MMM: A WWW-based method management system for using software modules remotely. *IEEE Internet Computing*. In press.

Günther, O. and H. Noltemeier (1991). Spatial database indices for large extended objects. In *Proc. 7th IEEE Int. Conf. on Data Eng.*, pp. 520–526.

Güting, R. H. (1989). Gral: An extendible relational datebase system for geometric applications. In *Proc. 15th Int. Conf. on Very Large Data Bases*, pp. 33–44.

Güting, R. H. and M. Schneider (1993). Realms: A foundation for spatial data types in database systems. In D. Abel and B. C. Ooi (Eds.), *Advances in Spatial Databases*, Number 692 in LNCS, Berlin/Heidelberg/New York. Springer-Verlag.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 47–54.

Hellerstein, J. M., E. Koutsoupias, and C. H. Papadimitriou (1997). Towards a theory of indexability. In *Proc. 16th ACM SIGACT–SIGMOD–SIGART Symp. on Principles of Database Systems*.

Hellerstein, J. M., J. F. Naughton, and A. Pfeffer (1995). Generalized search trees for database systems. In *Proc. 21th Int. Conf. on Very Large Data Bases*, pp. 562–573.

Henrich, A. (1995). Adapting the Transformation Technique to Maintain Multi-Dimensional Non-Point Objects in *k-d*-Tree Based Access Structures. In *Proc. 3rd ACM Int. Workshop on Advances in Geographic Information Systems (ACM-GIS'95)*, Baltimore, Maryland, USA. ACM Press.

Henrich, A. and J. Möller (1995). Extending a spatial access structure to support additional standard attributes. In M. J. Egenhofer and J. R. Herring (Eds.), *Proc. 4th Int. Symposium on Advances in Spatial Databases (SSD'95)*, Volume 951 of *LNCS*, pp. 132–151. Springer.

Henrich, A. and H.-W. Six (1991). How to split buckets in spatial data structures. In G. Gambosi, M. Scholl, and H.-W. Six (Eds.), *Geographic Database Management Systems*, pp. 212–244. Berlin/Heidelberg/New York: Springer-Verlag.

Henrich, A., H.-W. Six, and P. Widmayer (1989). The LSD tree: Spatial access to multidimensional point and non-point objects. In *Proc. 15th Int. Conf. on Very Large Data Bases*, pp. 45–53.

Hinrichs, K. (1985). Implementation of the grid file: Design concepts and experience. *BIT 25*, 569–592.

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.

Hoel, E. G. and H. Samet (1995). Benchmarking spatial join operations with spatial output. In *Proc. 21st Int. Conf. on Very Large Data Bases*, pp. 606–618.

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.

Hutflesz, A., H.-W. Six, and P. Widmayer (1988b). Twin grid files: Space optimizing access schemes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 183–190.

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.

Hutflesz, A., P. Widmayer, and C. Zimmermann (1991). Global order makes spatial access faster. In G. Gambosi, M. Scholl, and H.-W. Six (Eds.), *Geographic Database Management Systems*, pp. 161–176. Berlin/Heidelberg/New York: Springer-Verlag.

Informix Inc. (1997). The DataBlade architecture. URL http://www.informix.com.

Jagadish, H. V. (1990a). Linear clustering of objects with multiple attributes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 332–342.

Jagadish, H. V. (1990b). On indexing line segments. In *Proc. 16th Int. Conf. on Very Large Data Bases*, pp. 614–625.

Jagadish, H. V. (1990c). Spatial search with polyhedra. In *Proc. 6th IEEE Int. Conf. on Data Eng.*, pp. 311–319.

Kamel, I. and C. Faloutsos (1992). Parallel R-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 195–204.

Kamel, I. and C. Faloutsos (1993). On packing R-trees. In *Proc. 2nd Int. Conf. on Information and Knowledge Management*, pp. 490–499.

Kamel, I. and C. Faloutsos (1994). Hilbert R-tree: An improved R-tree using fractals. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pp. 500–509.

Kamel, I., M. Khalil, and V. Kouramajian (1996). Bulk insertion in dynamic R-trees. In *Proc. 7th Int. Symp. on Spatial Data Handling*, Delft, The Netherlands., pp. 3B.31–3B.42.

Kanellakis, P. C., S. Ramaswamy, D. E. Vengroff, and J. S. Vitter (1993). Indexing for data models with constraints and classes. In *Proc. 12th ACM SIGACT–SIGMOD–SIGART Symp. on Principles of Database Systems*, pp. 233–243.

Kemper, A. and M. Wallrath (1987). An analysis of geometric modeling in database systems. *ACM Computing Surveys 19*(1), 47–91.

Klinger, A. (1971). Pattern and search statistics. In S. Rustagi (Ed.), *Optimizing Methods in Statistics*, pp. 303–337.

Knott, G. (1975). Hashing functions. *The Computer Journal 18*(3), 265–278.

Kolovson, C. (1990). *Indexing Techniques for Multi-Dimensional Spatial Data and Historical Data in Database Management Systems*. Ph. D. thesis, University of California at Berkeley.

Kolovson, C. and M. Stonebraker (1991). Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 138–147.

Kriegel, H.-P. (1984). Performance comparison of index structures for multikey retrieval. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 186–196.

Kriegel, H.-P., P. Heep, S. Heep, M. Schiwietz, and R. Schneider (1991). An access method based query processor for spatial database systems. In G. Gambosi, M. Scholl, and H.-W. Six (Eds.), *Geographic Database Management Systems*, pp. 273–292. Berlin/Heidelberg/New York: Springer-Verlag.

Kriegel, H.-P., M. Schiwietz, R. Schneider, and B. Seeger (1990). Performance comparison of point and spatial access methods. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang (Eds.), *Design and Implementation of Large Spatial Database Systems*, Number 409 in LNCS, Berlin/Heidelberg/New York, pp. 89–114. Springer–Verlag.

Kriegel, H.-P. and B. Seeger (1986). Multidimensional order preserving linear hashing with partial expansions. In *Proc. Int. Conf. on Database Theory*, Number 243 in LNCS, Berlin/Heidelberg/New York. Springer–Verlag.

Kriegel, H.-P. and B. Seeger (1987). Multidimensional quantile hashing is very efficient for non–uniform record distributions. In *Proc. 3rd IEEE Int. Conf. on Data Eng.*, pp. 10–17.

Kriegel, H.-P. and B. Seeger (1988). PLOP–hashing: A grid file without directory. In *Proc. 4th IEEE Int. Conf. on Data Eng.*, pp. 369–376.

Kriegel, H.-P. and B. Seeger (1989). Multidimensional quantile hashing is very efficient for non–uniform distributions. *Information Sciences 48*, 99–117.

Kronacker, M. and D. Banks (1995). High-concurrency locking in R-trees. In *Proc. 21th Int. Conf. on Very Large Data Bases*, pp. 134–145.

Kumar, A. (1994a). G-tree: A new data structure for organizing multidimensional data. *IEEE Trans. Knowledge and Data Eng. 6*(2), 341–347.

Kumar, A. (1994b). A study of spatial clustering techniques. In D. Karagiannis (Ed.), *Proc. 5th Conf. on Database and Expert Systems Applications (DEXA'94)*, Number 856 in LNCS, pp. 57–70. Springer–Verlag.

Larson, P. A. (1980). Linear hashing with partial expansions. In *Proc. 6th Int. Conf. on Very Large Data Bases*, pp. 224–232.

Lehman, P. and S. Yao (1981). Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Systems 6*(4), 650–670.

Lin, K.-I., H. Jagadish, and C. Faloutsos (1994). The TV-tree: An index structure for high-dimensional data. *The VLDB J. 3*(4), 517–543.

Litwin, W. (1980). Linear hashing: A new tool for file and table addressing. In *Proc. 6th Int. Conf. on Very Large Data Bases*, pp. 212–223.

Lo, M. and C. Ravishankar (1994). Spatial joins using seeded trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 209–220.

Lomet, D. B. (1983). Boundex index exponential hashing. *ACM Trans. Database Systems 8*(1), 136–165.

Lomet, D. B. (1991). Grow and post index trees: Role, techniques and future potential. In O. Günther and H. Schek (Eds.), *Proc. 2nd Int. Symposium on Spatial Databases (SSD'91)*, Number 525 in LNCS, Berlin/Heidelberg/New York, pp. 183–206. Springer–Verlag.

Lomet, D. B. and B. Salzberg (1989). The hB–tree: A robust multiattribute search structure. In *Proc. 5th IEEE Int. Conf. on Data Eng.*, pp. 296–304.

Lomet, D. B. and B. Salzberg (1990). The hB–tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Systems 15*(4), 625–658. Reprinted in (Stonebraker 1994).

Lomet, D. B. and B. Salzberg (1992). Access method concurrency with recovery. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 351–360.

Lu, H. and B.-C. Ooi (1993). Spatial indexing: Past and future. *IEEE Data Eng. Bulletin 16*(3), 16–21.

Matsuyama, T., L. V. Hao, and M. Nagao (1984). A file organization for geographic information systems based on spatial proximity. *Int. J. Comp. Vision, Graphics and Image Processing 26*(3), 303–318.

Morton, G. (1966). A computer oriented geodetic data base and a new technique in file sequencing. IBM Ltd.

Nelson, R. and H. Samet (1987). A population analysis for hierachical data structures. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 270–277.

Ng, R. T. and J. Han (1994). Efficient and effective clustering methods for spatial data mining. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pp. 144–154.

Ng, V. and T. Kameda (1993). Concurrent accesses to R-trees. In D. Abel and B. C. Ooi (Eds.), *Advances in Spatial Databases*, Number 692 in LNCS, Berlin/Heidelberg/New York, pp. 142–161. Springer-Verlag.

Ng, V. and T. Kameda (1994). The R-link tree: A recoverable index structure for spatial data. In D. Karagiannis (Ed.), *Proc. 5th Conf. on Database and Expert Systems Applications (DEXA'94)*, Number 856 in LNCS, pp. 163–172. Springer–Verlag.

Nievergelt, J. (1989). 7±2 criteria for assessing and comparing spatial data structures. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang (Eds.), *Design and Implementation of Large Spatial Database Systems*, Number 409 in LNCS, Berlin/Heidelberg/New York, pp. 3–27. Springer–Verlag.

Nievergelt, J. and K. Hinrichs (1987). Storage and access structures for geometric data bases. In S. Ghosh, Y. Kambayashi, and K. Tanaka (Eds.), *Proc. Int. Conf. on Foundations of Data Organization 1985*, New York. Plenum Press.

Nievergelt, J., H. Hinterberger, and K. Sevcik (1981). 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.

Nievergelt, J., H. Hinterberger, and K. C. Sevcik (1984). The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Systems 9*(1), 38–71.

Ohsawa, Y. and M. Sakauchi (1983). BD-tree: A new n-dimensional data structure with efficient dynamic characteristics. In *Proc. 9th World Computer Congress, IFIP 1983*, pp. 539–544.

Ohsawa, Y. and M. Sakauchi (1990). A new tree type data structure with homogeneous node suitable for a very large spatial database. In *Proc. 6th IEEE Int. Conf. on Data Eng.*, pp. 296–303.

Ooi, B. C. (1990). *Efficient Query Processing in Geographic Information Systems.* Number 471 in LNCS. Berlin/Heidelberg/New York: Springer-Verlag.

Ooi, B. C., R. Sacks-Davis, and K. J. McDonell (1987). Spatial k-d-tree: An indexing mechanism for spatial databases. In *Proc. IEEE COMPSAC Conf.*

Ooi, B. C., R. Sacks-Davis, and K. J. McDonell (1991). Spatial indexing by binary decomposition and spatial bounding. *Information Systems J. 16*(2), 211–237.

Oosterom, P. (1990). *Reactive Data Structures for Geographic Information Systems.* Ph. D. thesis, University of Leiden, The Netherlands.

Oracle Inc. (1995). Oracle 7 multidimension: Advances in relational database technology for spatial data management. White Paper.

Orenstein, J. (1982). Multidimensional tries used for associative searching. *Information Processing Letters 14*(4), 150–157.

Orenstein, J. (1983). A dynamic file for random and sequential accessing. In *Proc. 9th Int. Conf. on Very Large Data Bases*, pp. 132–141.

Orenstein, J. (1989a). Redundancy in spatial databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 294–305.

Orenstein, J. (1989b). Strategies for optimizing the use of redundancy in spatial databases. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang (Eds.), *Design and Implementation of Large Spatial Database Systems*, Number 409 in LNCS, Berlin/Heidelberg/New York, pp. 115–134. Springer–Verlag.

Orenstein, J. (1990). A comparison of spatial query processing techniques for native and parameter space. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 343–352.

Orenstein, J. and T. H. Merrett (1984). A class of data structures for associative searching. In *Proc. 3rd ACM SIGACT–SIGMOD Symp. on Principles of Database Systems*, pp. 181–190.

Orenstein, J. A. (1986). Spatial query processing in an object-oriented database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 326–333.

Otoo, E. J. (1984). A mapping function for the directory of a multidimensional extendible hashing. In *Proc. 10th Int. Conf. on Very Large Data Bases*, pp. 493–506.

Otoo, E. J. (1985). Symmetric dynamic index maintenance scheme. In *Proc. Int. Conf. on Foundations of Data Organization*, pp. 283–296.

Otoo, E. J. (1986). Balanced multidimensional extendible hash tree. In *Proc. 5th ACM SIGACT–SIGMOD Symp. on Principles of Database Systems*, pp. 100–113.

Ouksel, M. (1985). The interpolation based grid file. In *Proc. 4th ACM SIGACT–SIGMOD Symp. on Principles of Database Systems*, pp. 20–27.

Ouksel, M. and P. Scheuermann (1983). Storage mappings for multidimensional linear dynamic hashing. In *Proc. 2th ACM SIGACT–SIGMOD Symp. on Principles of Database Systems*, pp. 90–105.

Ouksel, M. A. and O. Mayer (1992). A robust and efficient spatial data structure. *Acta Informatica 29*, 335–373.

Overmars, M. H., M. H. Smid, T. Berg, and M. J. van Kreveld (1990). Maintaining range trees in secondary memory: Part I: Partitions. *Acta Informatica 27*, 423–452.

Pagel, B. U., H.-W. Six, and H. Toben (1993). The transformation technique for spatial objects revisited. In D. Abel and B. C. Ooi (Eds.), *Advances in Spatial Databases*, Number 692 in LNCS, Berlin/Heidelberg/New York, pp. 73–88. Springer-Verlag.

Pagel, B. U., H.-W. Six, H. Toben, and P. Widmayer (1993). Towards an analysis of range query performance in spatial data structures. In *Proc. 12th ACM SIGACT-SIGMOD–SIGART Symp. on Principles of Database Systems*, pp. 214–221.

Pagel, B. U., H.-W. Six, and M. Winter (1995). Window query optimal clustering of spatial objects. In *Proc. 14th ACM SIGACT-SIGMOD–SIGART Symp. on Principles of Database Systems*, pp. 86–94.

Papadias, D., Y. Theodoridis, T. Sellis, and M. J. Egenhofer (1995). Topological relations in the world of minimum bounding rectangles: A study with R-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 92–103.

Papadopoulos, A. and Y. Manolopoulos (1997). Performance of nearest neighbor queries in R-trees. In F. Afrati and P. Kolaitis (Eds.), *Proc. Int. Conf. on Database Theory (ICDT'97)*, Number 1186 in LNCS, Berlin/Heidelberg/New York, pp. 394–408.

Peloux, J., G. Reynal, and M. Scholl (1994). Evaluation of spatial indices implemented with the O_2 DBMS. *Ingénièrie des systèmes d'information 6.*

Preparata, F. P. and M. I. Shamos (1985). *Computational geometry*. New York, NY: Springer–Verlag.

Regnier, M. (1985). Analysis of the grid file algorithms. *BIT 25*, 335–357.

Robinson, J. T. (1981). The K–D–B–tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 10–18.

Rotem, D. (1991). Spatial join indices. In *Proc. 7th IEEE Int. Conf. on Data Eng.*, pp. 10–18.

Roussopoulos, N. and D. Leifker (1984). An introduction to PSQL: A pictoral structured query language. In *Proc. IEEE Workshop on Visual Languages*.

Roussopoulos, N. and D. Leifker (1985). Direct spatial search on pictorial databases using packed R–trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 17–31.

Sagan, H. (1994). *Space-Filling Curves*. Berlin/Heidelberg/New York: Springer – Verlag.

Samet, H. (1984). The quadtree and related hierarchical data structure. *ACM Computing Surveys 16*(2), 187–260.

Samet, H. (1988). Hierarchical representation of collections of small rectangles. *ACM Computing Surveys 20*(4), 271–309.

Samet, H. (1989). *The design and analysis of spatial data structures*. Reading, MA: Addison–Wesley.

Samet, H. (1990). *Applications of Spatial Data Structures*. Reading, MA: Addison–Wesley.

Samet, H. and R. E. Webber (1985). Storing a collection of polygons using quadtrees. *ACM Trans. Graphics 4*(3), 182–222.

Schiwietz, M. (1993). *Speicherung und Anfragebearbeitung komplexer Geo-Objekte*. Ph. D. thesis, Ludwig-Maximilians-Universität München, Germany. In German.

Schneider, R. and H.-P. Kriegel (1992). The TR*-tree: A new representation of polygonal objects supporting spatial queries and operations. In *Proc. 7th Workshop on Computational Geometry*, Number 553 in LNCS, Berlin/Heidelberg/New York, pp. 249–264. Springer-Verlag.

Scholl, M. and A. Voisard (1989). Thematic map modeling. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang (Eds.), *Design and Implementation of Large Spatial Database Systems*, Number 409 in LNCS, Berlin/Heidelberg/New York. Springer–Verlag.

Seeger, B. (1991). Performance comparison of segment access methods implemented on top of the buddy-tree. In *Advances in Spatial Databases*, Number 525 in LNCS, Berlin/Heidelberg/New York, pp. 277–296. Springer–Verlag.

Seeger, B. and H.-P. Kriegel (1988). Techniques for design and implementation of spatial access methods. In *Proc. 14th Int. Conf. on Very Large Data Bases*, pp. 360–371.

Seeger, B. and H.-P. Kriegel (1990). The buddy–tree: An efficient and robust access method for spatial data base systems. In *Proc 16th Int. Conf. on Very Large Data Bases*, pp. 590–601.

Sellis, T., N. Roussopoulos, and C. Faloutsos (1987). The R^+-tree: A dynamic index for multi-dimensional objects. In *Proc. 13th Int. Conf. on Very Large Data Bases*, pp. 507–518.

Sevcik, K. and N. Koudas (1996). Filter trees for managing spatial data over a range of size granularties. In *Proc. 22th Int. Conf. on Very Large Data Bases*, Bombay, India, pp. 16–27.

Sexton, A. P. (1997). Querying indexed files. In V. Gaede, A. Brodsky, O. Günther, V. Vianu, and M. Wallace (Eds.), *Proc. CDB'97 and CP'96 Workshop on Constraint Databases and their Application*, Number 1191 in LNCS, Berlin/Heidelberg/New York, pp. 263–281. Springer–Verlag.

Shekhar, S. and D.-R. Liu (1995). CCAM: A connectivity-clustered access method for aggregate queries on transportation networks: A summary of results. In *Proc. 11th IEEE Int. Conf. on Data Eng.*, pp. 410–419.

Six, H. and P. Widmayer (1988). Spatial searching in geometric databases. In *Proc. 4th IEEE Int. Conf. on Data Eng.*, pp. 496–503.

Smid, M. H. and M. H. Overmars (1990). Maintaining range trees in secondary memory part II: Lower bounds. *Acta Informatica 27*, 453–480.

Smith, T. R. and P. Gao (1990). Experimental performance evaluations on spatial access methods. In *Proc. 4th Int. Symp. on Spatial Data Handling*, Zürich, pp. 991–1002.

Stonebraker, M. (Ed.) (1994). *Readings in Database Systems.* San Mateo: Morgan Kaufmann.

Stonebraker, M., T. Sellis, and E. Hanson (1986). An analysis of rule indexing implementations in data base systems. In *Proc. 1st Int. Conf. on Expert Data Base Systems.*

Stuckey, P. (1997). Constraint search trees. In L. Naish (Ed.), *Proc. Int. Conf. on Logic Programming (CLP'97).* MIT Press.

Subramanian, S. and S. Ramaswamy (1995). The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA'95).*

Tamminen, M. (1982). The extendible cell method for closest point problems. *BIT 22*, 27–41.

Tamminen, M. (1983). Performance analysis of cell based geometric file organisations. *Int. J. Comp. Vision, Graphics and Image Processing 24*, 160–181.

Tamminen, M. (1984). Comment on quad- and octrees. *Communications of the ACM 30*(3), 204–212.

Theodoridis, Y. and T. K. Sellis (1996). A model for the prediction of R-tree performance. In *Proc. 15th ACM SIGACT–SIGMOD–SIGART Symp. on Principles of Database Systems*, pp. 161–171.

Tropf, H. and H. Herzog (1981). Multidimensional range search in dynamically balanced trees. *Angewandte Informatik 2*, 71–77.

Whang, K.-Y. and R. Krishnamurthy (1985). *Multilevel grid files*. Yorktown Heights, NY: IBM Research Laboratory.

White, M. (1981). N-trees: Large ordered indexes for multi-dimensional space. Technical report, Application Mathematics Research Staff, Statistical Research Division, US Bureau of the Census.

Widmayer, P. (1991). Datenstrukturen für Geodatenbanken. In G. Vossen and K.-U. Witt (Eds.), *Entwicklungstendenzen bei Datenbank-Systemen*, Chapter 9, pp. 317–361. Munich: Oldenbourg–Verlag. In German.