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*, which are able to manage extended objects (such as rectangles or polyhedra). We conclude with a discussion of theoretical and experimental results concerning the relative performance of the various approaches.

Keywords: access methods, data structures, spatial databases

Contents

1 Introduction 4

2 Organization of Spatial Data 6
- 2.1 What Is Special About Spatial? 6
- 2.2 Definitions and Queries 8

3 Basic Data Structures 12
- 3.1 One-Dimensional Access Methods 12
  - 3.1.1 Linear Hashing 12
  - 3.1.2 Extendible Hashing 13
  - 3.1.3 The B-Tree 13
- 3.2 Main Memory Structures 13
  - 3.2.1 The K-D-Tree 14
  - 3.2.2 The BSP Tree 16
  - 3.2.3 The Quadtree 17

4 Point Access Methods 18
- 4.1 Multidimensional Hashing 19
  - 4.1.1 The Grid File 19
  - 4.1.2 EXCELL 22
  - 4.1.3 The Two-Level Grid File 22
  - 4.1.4 The BANG File 22
  - 4.1.5 The Twin Grid File 23
  - 4.1.6 Multidimensional Linear Hashing 25
  - 4.1.7 The Buddy Tree 26
- 4.2 Hierarchical Access Methods 27
  - 4.2.1 The K-D-B-Tree 27
  - 4.2.2 The hB-Tree 29
  - 4.2.3 The LSD Tree 30

5 Spatial Access Methods 31
- 5.1 Managing Extended Objects 31
  - 5.1.1 Transformation 32
  - 5.1.2 Overlapping Regions 33
  - 5.1.3 Clipping 34
- 5.2 Space-Filling Curves 35
  - 5.2.1 Z-Ordering 36
- 5.3 The R-Tree and Related Structures 38
  - 5.3.1 The R-Tree 38
  - 5.3.2 The R^+-Tree 40
  - 5.3.3 The R*-Tree 41
- 5.4 Polyhedral Tree Structures 42
  - 5.4.1 The Cell Tree 42
  - 5.4.2 The P-Tree (Jagadish 1990c) 44

5.4.3 The P-Tree (Schiwietz 1993) 45
5.4.4 The BV-Tree 46
5.5 Other Hierarchical Access Methods 47
5.5.1 The Extended K-D-Tree 47
5.5.2 The SKD-Tree 48
5.5.3 The GBD-Tree 49
5.6 Hashing-Based Structures 50
5.6.1 PLOP-Hashing 50
5.6.2 The Multi-Layer Grid File 52
5.6.3 The R-File 52

6 Comparative Studies **54**

7 Conclusions **57**

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 in the geosciences and mechanical CAD, the range of possible applications has now 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 operators. Cartographic data is also two–dimensional with points, lines, and regions as the 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 excellent overview of work published before 1991. Like the thesis by Schiwietz, this survey is however only available in German.

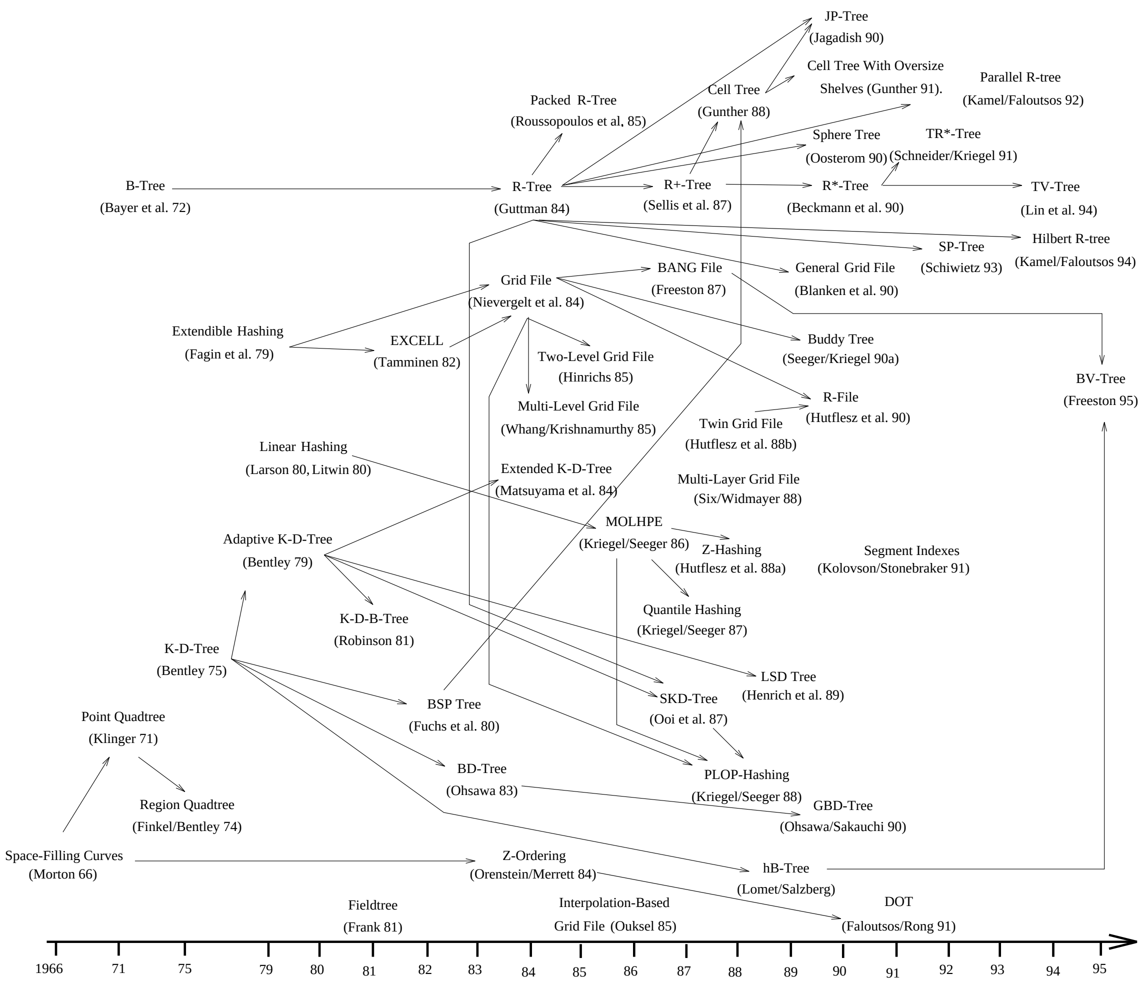

Figure 1: History of Multi-Dimensional Access Methods

The sequel 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 are 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 analytical results 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, let us 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 point sets, 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. Third, spatial databases tend to be *large.* Geographic maps, for example, regularly occupy several gigabytes of storage. The seamless 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. 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, might 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 between operators, spatial database operators 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. Given a set of geometric objects in d–dimensional Euclidean space E^d, stored in a spatial database, a *region query* computes those objects in the database that overlap a given search space $S \subset E^d$. The *point query* can be viewed as a degenerate region query, determining all objects containing the given search point $p \in E^d$. 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 natural approach to handle multidimensional search queries consists in the consecutive application of such single key structures, one per dimension. As Kriegel (1984) has pointed out, however, this approach can be quite inefficient. Since each index is traversed independently of the others, we can not exploit the possibly high selectivity in one dimension for narrowing down the search in the remaining dimensions. Another interesting approach is to extend hashing simply by using a hash function that takes a d-dimensional vector as argument. A structure that is based on this idea is the grid file (Nievergelt, Hinterberger, and Sevcik 1984; Hinrichs 1985). Unfortunately this approach often suffers from a superlinear growth of the directory, even for data that is uniformly distributed; in Sections 3 and 4 we provide a more detailed discussion of these and related problems.

In summary, there is no easy and obvious way to extend single key structures in order to handle multidimensional data. On the other hand, there is a great 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):

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. Access methods therefore 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.* Access methods should maintain their efficiency even when the input data is highly skewed. This point is especially important for data that is distributed differently along the various dimensions. Related to this point is a good average storage utilization.

5. *Simplicity.* Intricate access methods with special cases are often error-prone to implement and thus not sufficiently robust.

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

7. *Efficiency.* Because spatial searches are usually CPU- *and* I/O-intensive, an access method should try to minimize both the number of disk accesses and the CPU utilization.

As for terminology, 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. The points may have 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.*

2.2 Definitions and Queries

As indicated above, we generally assume that the given objects are embedded in d-dimensional Euclidean space E^d. In this paper this space is also referred to as *universe* or *original space*. The restriction to Euclidean space is made for simplicity and practicality; it is not always essential, i.e., many observations in this paper also hold in non-Euclidean spaces.

Any point object stored in a spatial database has a unique location in space, 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 $1 \cdot H(a,c) = \{x \in E^d : x \cdot a \geq c\}$ and $-1 \cdot H(a,c) = \{x \in E^d : x \cdot a \leq c\}$. 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 asssume 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 its *geometry* $o.G$, which is a special attribute of some spatial data type.

Indices can typically only handle simple entries of roughly equal size. One therefore often abstracts from the actual geometry of a spatial object before inserting it into an index. This can be achieved by approximating the original data object with a simpler geometry, 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 description of the object geometry (the *object ID* or *object reference*). This geometry is retrieved from disk if and only if it is essential for answering a given query.

With this design, the index only produces a set of *candidate solutions* (Figure 2).

For each element of that candidate set we have to decide whether the MBB is sufficient to decide that the actual object *must* indeed satisfy 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 information. We then have to retrieve the exact geometries from secondary memory and test them against the predicate. If the predicate evaluates to true, the object is added to the query result as well, otherwise we have a *false hit*.

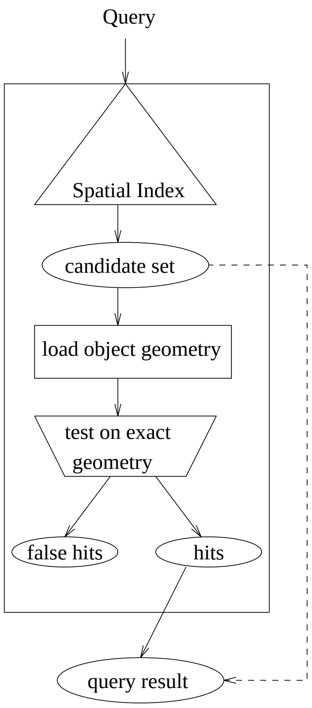

Figure 2: Multi-Step Spatial Query Processing (see also (Brinkhoff et al. 1994))

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 sequel, we give a formal definition of several of the more common spatial database operators.

Query 1 (Exact Match Query EMQ, Object Query) *Given an object o' with geometry $o'.G \subseteq E^d$, find all objects o with the same geometry 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 queries introduced in the sequel, 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 with 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 geometry $o'.G \subseteq E^d$, find all objects o with 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 geometry $o'.G \subseteq E^d$, 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 geometry $o'.G \subseteq E^d$, 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 inputs.

Query 7 (Adjacency Query AQ) *Given an object o' with geometry $o'.G \subseteq E^d$, 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 geometries $o'.G$ and $o.G$, respectively.

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

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

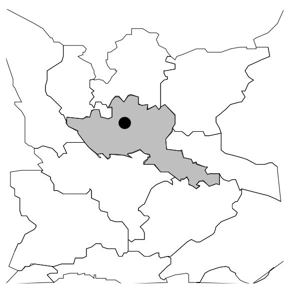

Figure 3: Point Query

Figure 4: Window Query

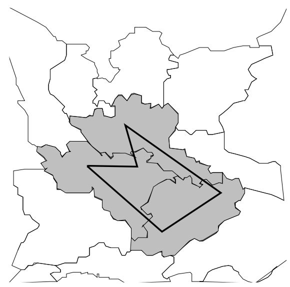

Figure 5: Intersection Query

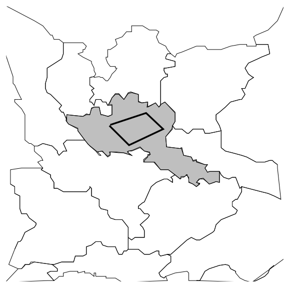

Figure 6: Enclosure Query

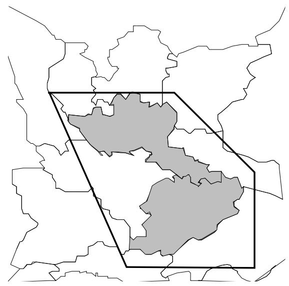

Figure 7: Containment Query

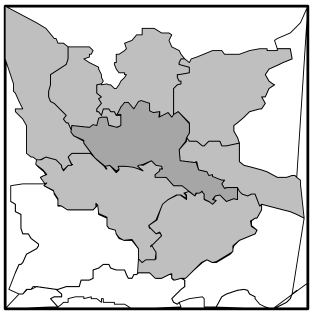

Figure 8: Adjacency Query

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.

Figures 3 through 8 illustrate some of the queries defined above.

Besides spatial selections, as exemplified by Queries 1 through 8, the *spatial join* is one of the most important spatial operations. Günther (1993) gives the following definition of the spatial join in a relational context:

Query 9 (Spatial Join) *The spatial join of two relations R and S, denoted by $R \bowtie_{i\theta j} S$, is the set of tuples from $R \times S$ where the i-th column of R and the j-th column of S are of some spatial type, θ is some spatial predicate, and $R.i$ stands in relation θ to $S.j$.*

Typically, one dedicated column in the relations R and S is of some spatial data type, representing the geometry of the corresponding data object. We will therefore just write $R \bowtie_\theta S$ as a shorthand for the spatial join $R \bowtie_{i\theta j} S$ whenever i and j refer to such dedicated columns in the relations R and S, respectively. This definition can be extended to an object-oriented environment in a straightforward manner by viewing the tuples $r \in R$ and $s \in S$ as (spatial) objects and the relations R and S as sets of objects or class extents.

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; Brinkhoff 1994;

Lo and Ravishankar 1994; Aref and Samet 1994) yields a wide variety of possibilities, including

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

A closer inspection of these spatial predicates shows that the *intersection join* $R \bowtie_{intersects(r,s)} S$ plays a crucial role for the computation in practically all 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 tuples (r, s) 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. Some of 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. This problem is aggravated by the use of *order-preserving* hash functions 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. For well-behaved data, on the other hand, hashing techniques usually outperform hierarchical methods on the average.

3.1.1 Linear Hashing (Larson 1980; Litwin 1980)

Linear hashing divides the universe $[A, B)$ of possible 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. 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 reaches its 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. Exact 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.

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 remaining nodes on 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), B-trees are usually rather adaptive. For data whose distribution is sufficiently close to uniform, however, extendible hashing as well as linear hashing outperform the B-tree.

3.2 Main Memory Structures

Early 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.

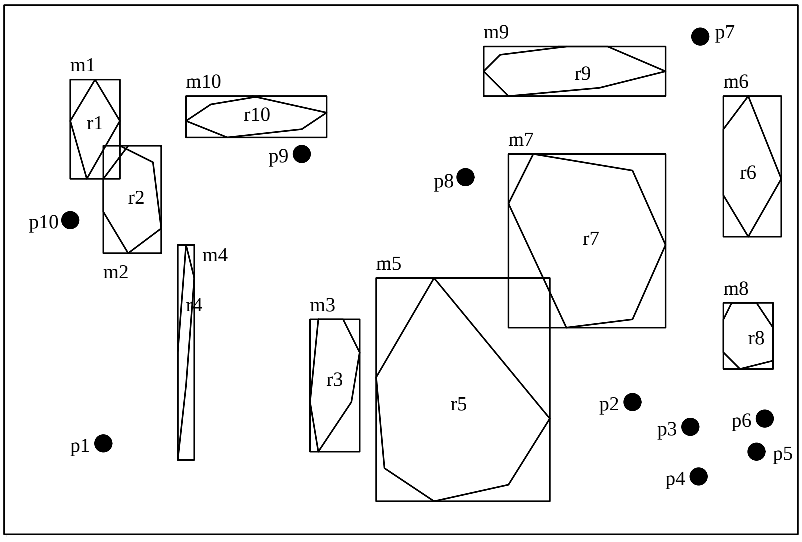

Figure 9: Running Example

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 the 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 parallel to the $x-y-$, the $x-z-$, and the $y-z-$plane (possibly in some other order). 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.

One disadvantage of the k-d-tree is that the structure depends on 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) solves these problems by choosing

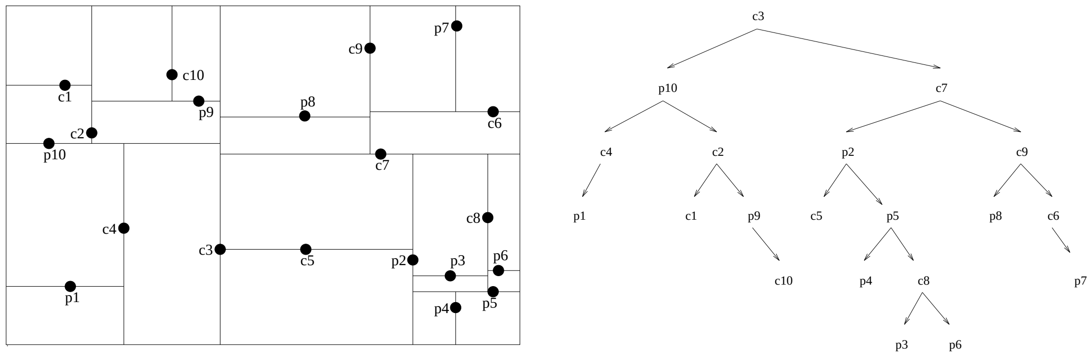

Figure 10: K-D-Tree

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 single point. The adaptive k-d-tree is not a very dynamic 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.

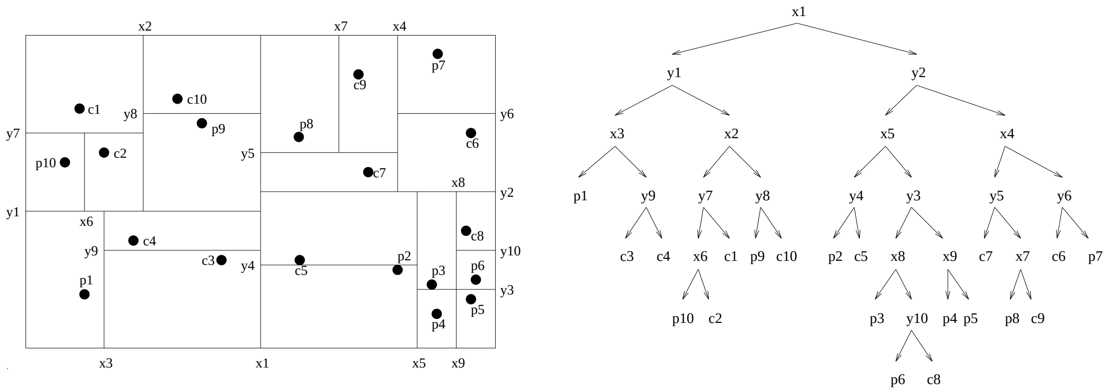

Figure 11: Adaptive K-D-Tree

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 single point. 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 text, we shall encounter several other structures that are based on this kind of partitioning.

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 points in a given subspace. The decomposition usually continues until the number of points 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 data points that are contained in the corresponding subspace. Figure 12 shows a BSP tree for the running example with no more than two data points per subspace.

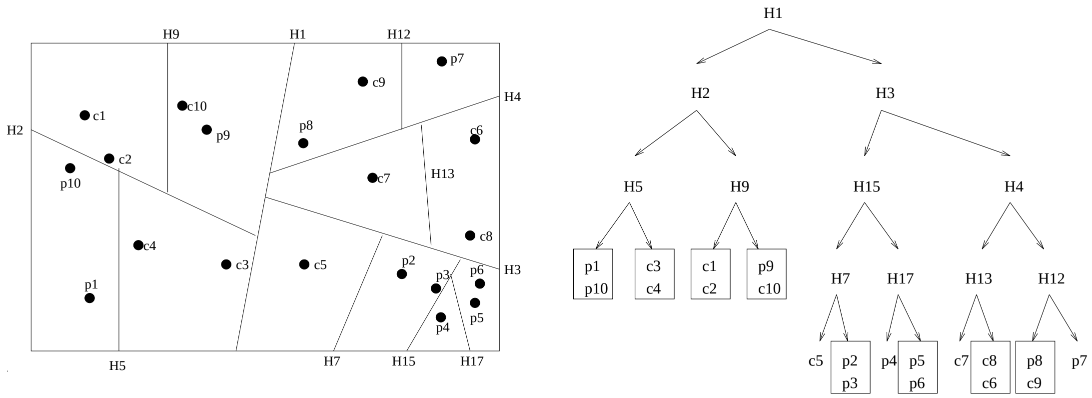

Figure 12: BSP Tree

In order to perform a point search, we insert the search point into the root of the tree and determine on which side of the corresponding hyperplane the point 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 points that are contained in the corresponding subspace whether they coincide with the search point. The range search algorithm is a straightforward generalization.

BSP trees can adapt well to the distribution of the data in the database. 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 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, only one subtree qualifies, whereas for range queries there may be 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 at 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 in the subtree. Figure 13 shows a two-dimensional point quadtree for the running example.

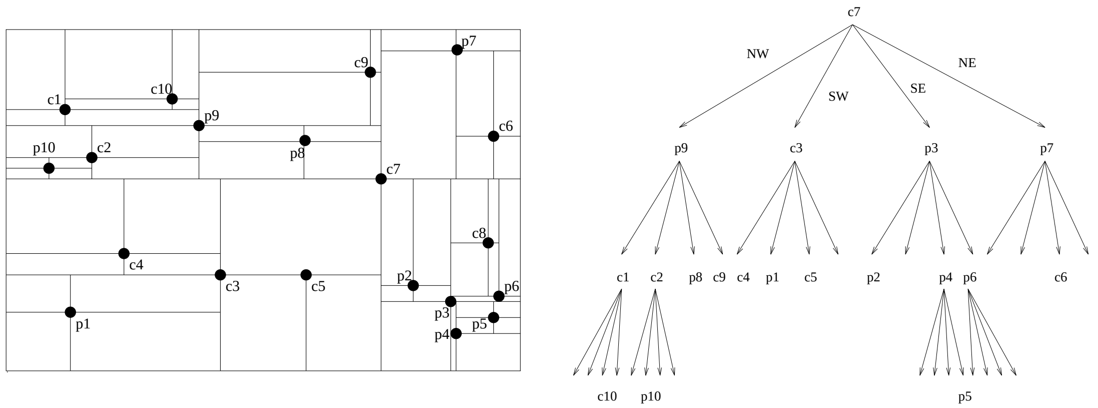

Figure 13: Point Quadtree

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. For the running example, Figure 14 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 and vertices. These edges and vertices (which together form an exact description of the actual 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.

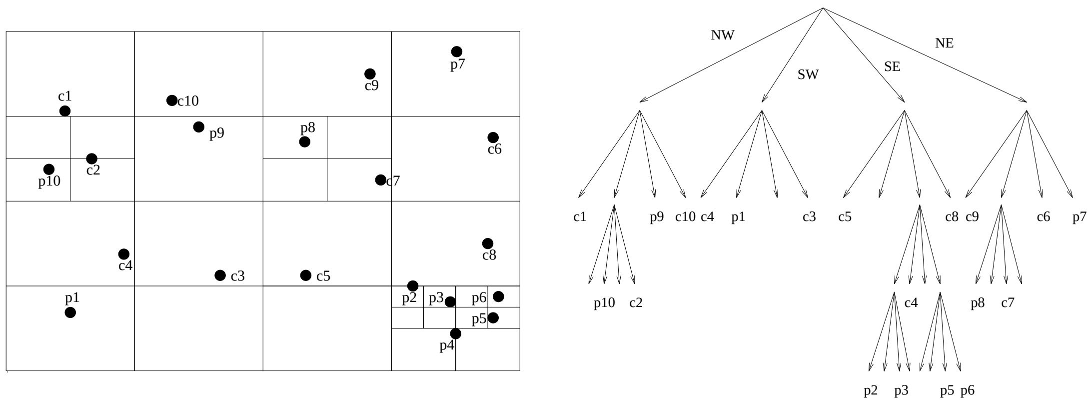

Figure 14: Region Quadtree

4 Point Access Methods

The multidimensional data structures presented in the previous section do not take secondary storage management explicitly into account. 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. Especially 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 *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 hashing scheme.

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; Freeston 1987), which organize the directory as a tree structure. Tree–based access methods are usually a generalization of the B–tree (Bayer and McCreight 1972) to higher dimensions, such as the k-d-B-tree (Robinson 1981) or the hB-tree (Lomet and Salzberg 1989).

In the sequel, 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. Seeger and Kriegel (1990a) propose another interesting taxonomy, which classifies point access methods according to 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 they 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.

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 in the hashing order at least to some extent. The goal of all these heuristics is that objects that are located close to each other should be stored close together on the disk with high probability. This could contribute substantially to minimizing the number of seek operations per range query. We begin our presentation with several multidimensional hashing structures based on extendible hashing. Structures based on linear hashing are discussed in Section 4.1.6, and 4.1.7 presents the buddy tree, a hybrid method.

4.1.1 The Grid File (Nievergelt, Hinterberger, and Sevcik 1984)

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 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, the grid itself is kept in main memory, represented by d one-dimensional arrays called *scales*. Average space occupancy of the grid file was proven to be around 69% (ln 2).

property			point access method
intervals	complete	disjoint	
×	×	×	quadtree (Finkel and Bentley 1974; Samet 1984), k–d–B tree (Robinson 1981), EXCELL (Tamminen 1982), multidimensional extendible hashing (Otoo 1984), interpolation hashing (Burkhard 1983), grid file (Nievergelt, Hinterberger, and Sevcik 1984), balanced multidimensional two–level grid file (Hinrichs 1985), interpolation–based grid file (Ouksel 1985), MOLHPE (Kriegel and Seeger 1986), extendible hash tree (Otoo 1985), 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 1990a)
	×	×	BSP tree (Fuchs, Kedem, and Naylor 1980), BANG file (Freeston 1987), hB–tree (Lomet and Salzberg 1989)

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

Figure 15 shows a grid file for the running example. We assume bucket capacity to be four data points. The center of the figure shows the root directory with scales on the x- and y-axes. The data points are pictures in this directory for demonstration purposes only, they are of course not stored there. In the lower left part, four cells are combined to a single bucket, represented by four pointers to a single page. Hence there are four directory entries for the same page, which illustrates a well-known problem of the grid file: the excessive growth of the directory (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 1990a). 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 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 the 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.

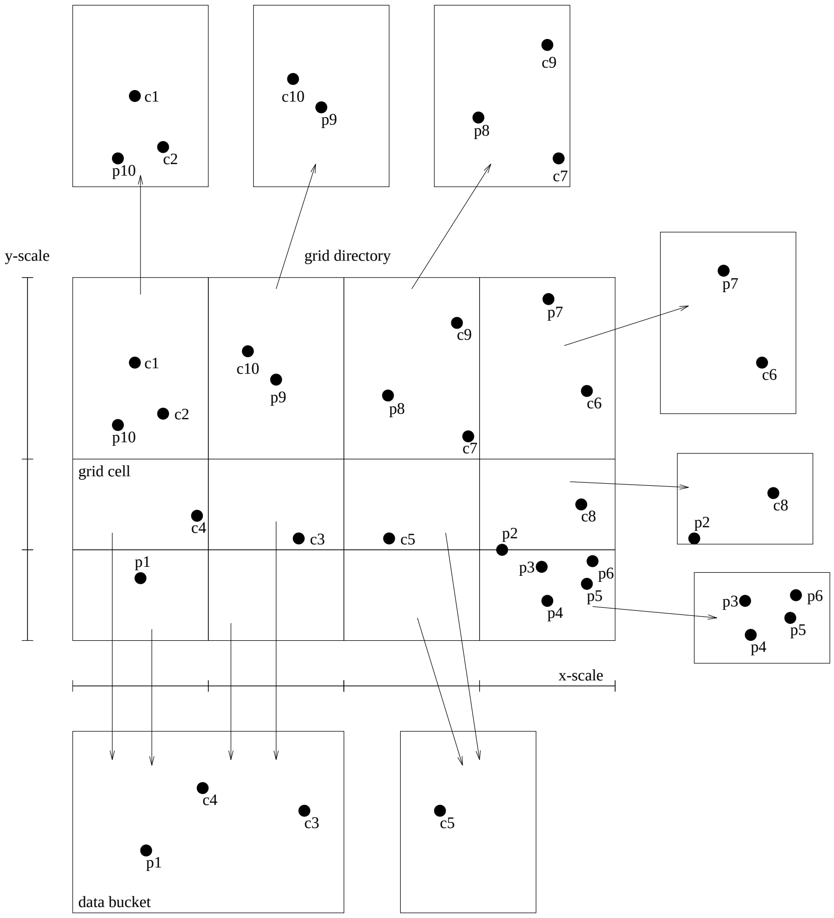

Figure 15: Grid File

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, 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 an exponential growth of the directory (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. To check for this requires a complete scan of the grid directory. Hinrichs (1985) 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 as long as 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*, where no merging is feasible because the resulting bucket region would not be box-shaped (Hinrichs 1985; Seeger and Kriegel 1990b). For a theoretical analysis of the grid file and some of its variants see (Regnier 1985) or (Becker 1992).

4.1.2 EXCELL (Tamminen 1982)

A close relative of 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 in a *regular* manner; 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 growth of the directory, it does not solve the problem completely. Figure 16 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 BANG File (Freeston 1987)

One of the first proposals addressing the exponential growth of the directory was the *interpolation-based grid file* (Ouksel 1985). Interpolation-based grid files avoid this particular problem 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.

To obtain a better adaption to the given data points, Freeston (1987) proposed a similar structure called the *BANG (Balanced And Nested Grid) file*. Like the grid file, the BANG file 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 the different

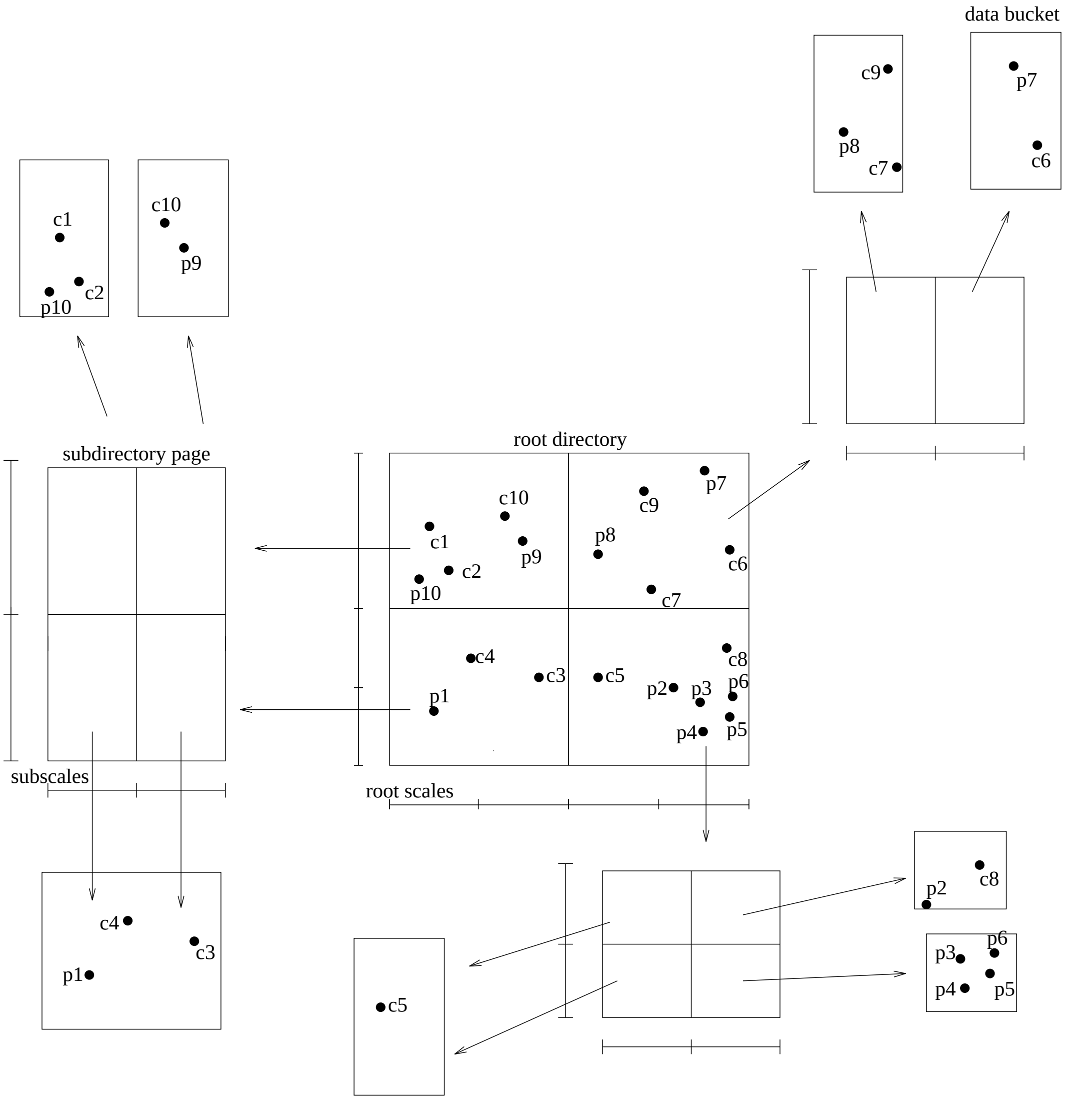

Figure 16: Two-Level Grid File

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 17 shows a 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.

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

The *twin grid file* tries to increase space utilization compared to the regular grid file by introducing a second grid file. As indicated by the name "twin," the relationship

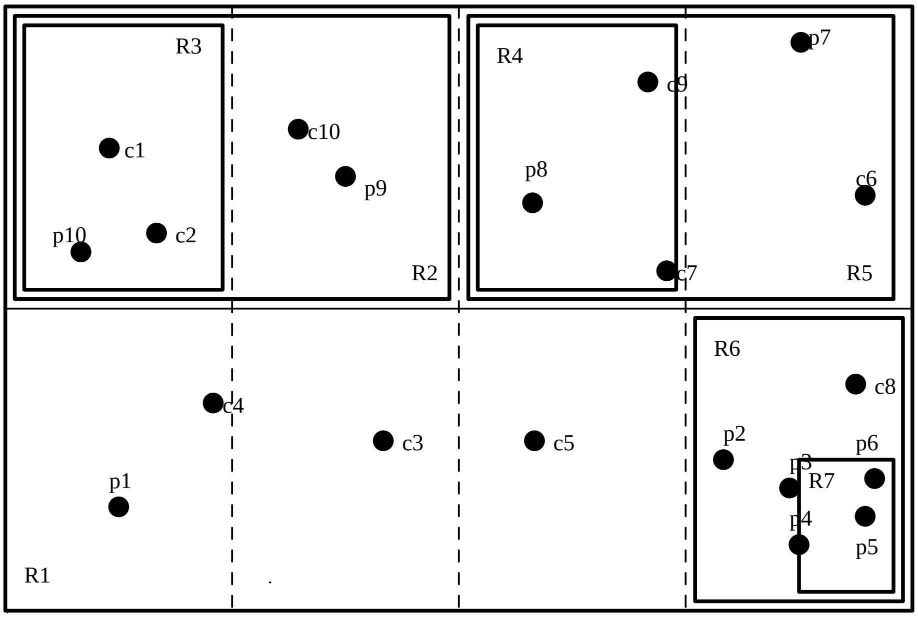

Figure 17: BANG File

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 are spanning the whole universe and 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 standard grid file) without experiencing substantial performance penalties.

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 to 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. It should be pointed out that there usually exists no unique optimum for the distribution of the data points among the two files.

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

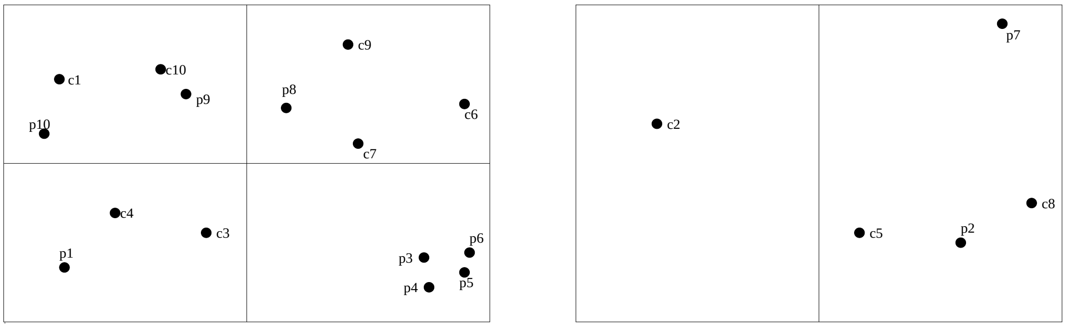

Figure 18: Twin Grid File

4.1.6 Multidimensional Linear Hashing

Unlike multidimensional extendible hashing, multidimensional *linear* hashing uses no or only a very small directory. It therefore uses 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 an expansion pointer $ep = (ep_1, ..., ep_d)$ 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 has been called *quantile hashing* (Kriegel and Seeger 1987; Kriegel and Seeger 1989). The critical property of the binary 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 5.6.1.

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 5.2.1. One disadvantage of z-hashing is that a number of useless data blocks will be generated, similar as in the case of 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 scope due to their inability to adapt to different data distributions.

4.1.7 The Buddy Tree (Seeger and Kriegel 1990a)

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 in (Seeger and Kriegel 1990a; Seeger and Kriegel 1990b) 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 19.

The interpolation-based grid file by Ouksel (1985) and the balanced multidimensional extendible hash tree by Otoo (1986) are both special cases of the buddy tree; they can be obtained by restricting the properties of the regions.

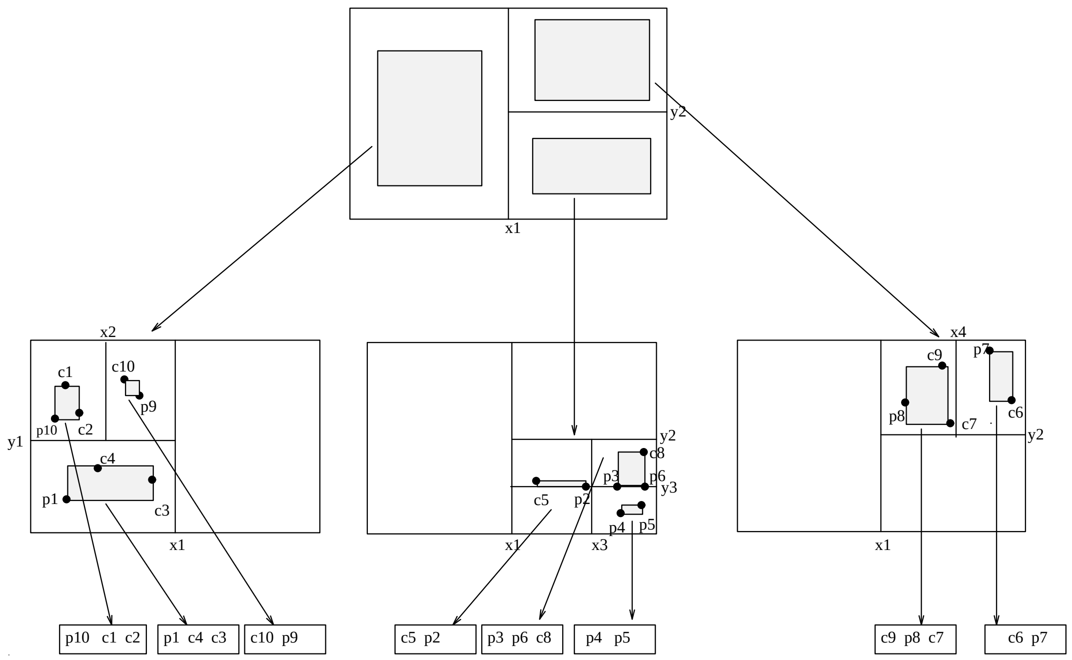

Figure 19: Buddy Tree

4.2 Hierarchical Access Methods

In this section we discuss several PAMs that are based on a binary or a multiway tree structure. Other than the buddy tree, which is a hybrid structure, 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 to 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.

Differences between tree structures are mainly based on the characteristics of the regions. Table 1 on page 20 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 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 on 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 20.

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 this data node 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.

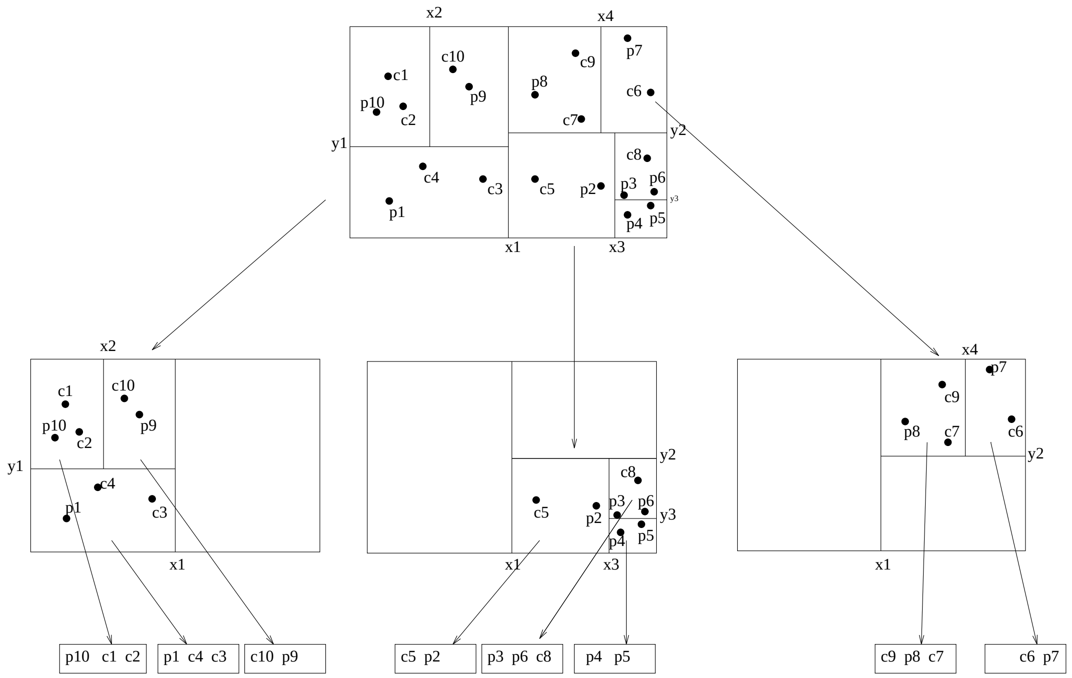

Figure 20: K-D-B-Tree

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

The *hB-tree* (holey brick tree) is a relative of 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 the splitting of the node is done 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, with this technique the cascading of splits that is typical for many other structures can be avoided.

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 geometries, this results in forming the union of the corresponding regions. As in the case of the BANG file, which is based on the *difference* of regions only, the resulting region may not be rectangular anymore. This peculiarity is illustrated in Figure 21, 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 $y1$.

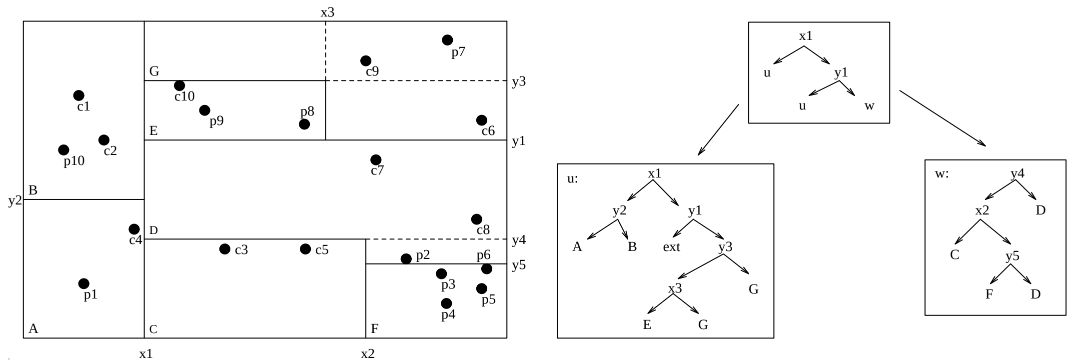

Figure 21: hB-Tree

Searching is similar to the k-d-B-tree; each internal k-d-tree is traversed as usual. The performance for one-dimensional searches is competitive with the B$^+$-tree. 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 a 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 $1/3 : 2/3$.

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, i.e., 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.

Although the hB-tree has several advantages compared to other structures, there are some features that may cause problems in certain applications. First, one may store redundant references to child nodes, which may lead to a *superlinear growth* of the index in the case of skewed data. Second, it seems rather complicated to delete objects and to perform the restructuring that may be required afterwards. No algorithm has been published so far; according to the inventors of the hB-tree, an algorithm exists, albeit complicated, since a number of cases need to be considered.

In a recent Ph.D. thesis (Evangelidis 1994), the hB-tree has been extended to allow for concurrency and recovery. The new structure is called the hB^{Π}-*tree*.

4.2.3 The LSD Tree (Henrich, Six, and Widmayer 1989)

We list the *LSD (Local Split Decision) tree* as a point access method although it has been emphasized by its authors 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 is height-balanced. 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. Figure 22 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 that the data is uniformly distributed. On the contrary, it tries to accomodate 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

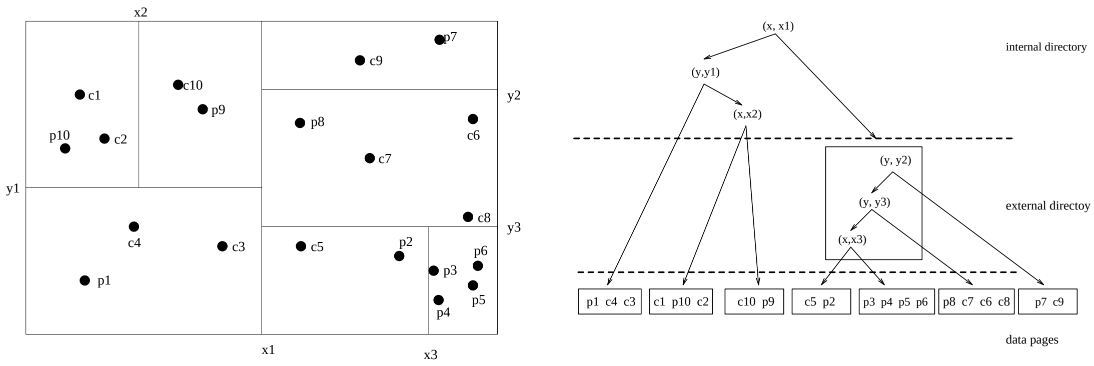

Figure 22: LSD Tree

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, we form 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.

5 Spatial Access Methods

5.1 Managing Extended Objects

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 three techniques (Seeger and Kriegel 1988; Ooi 1990), which we shall discuss in the sequel:

1. Transformation (Object Mapping)

2. Overlapping Regions (Object Bounding)

3. Clipping (Object Duplication)

5.1.1 Transformation

Objects with a simple geometric structure, such as circles or rectangles, can be represented as points in a higher-dimensional space (Hinrichs 1985; Seeger and Kriegel 1988). 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.

Figure 23 shows the dual space equivalents of some common queries. For simplicity of presentation 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 23a 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 23b). 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 23c gives an example with p as the (one-dimensional) query point.

Despite its conceptual elegance, this approach has several disadvantages. First, as the examples above have already indicated, the formulation of point and range queries in dual space is usually much more complicated than in the original space (Nievergelt and Hinrichs 1987). Finite search regions may map into infinite search regions in dual space, and some more complex queries (such as distance-based queries) 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. If we use a tree structure as PAM, these problems may cause serious performance penalties (Greene 1989). Non–hierarchical index structures, such as the grid file or EXCELL, seem to suffer less from these disadvantages (Nievergelt and Hinrichs 1987).

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 (Henrich et al. 1989) described in Section 4.2.3. 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,

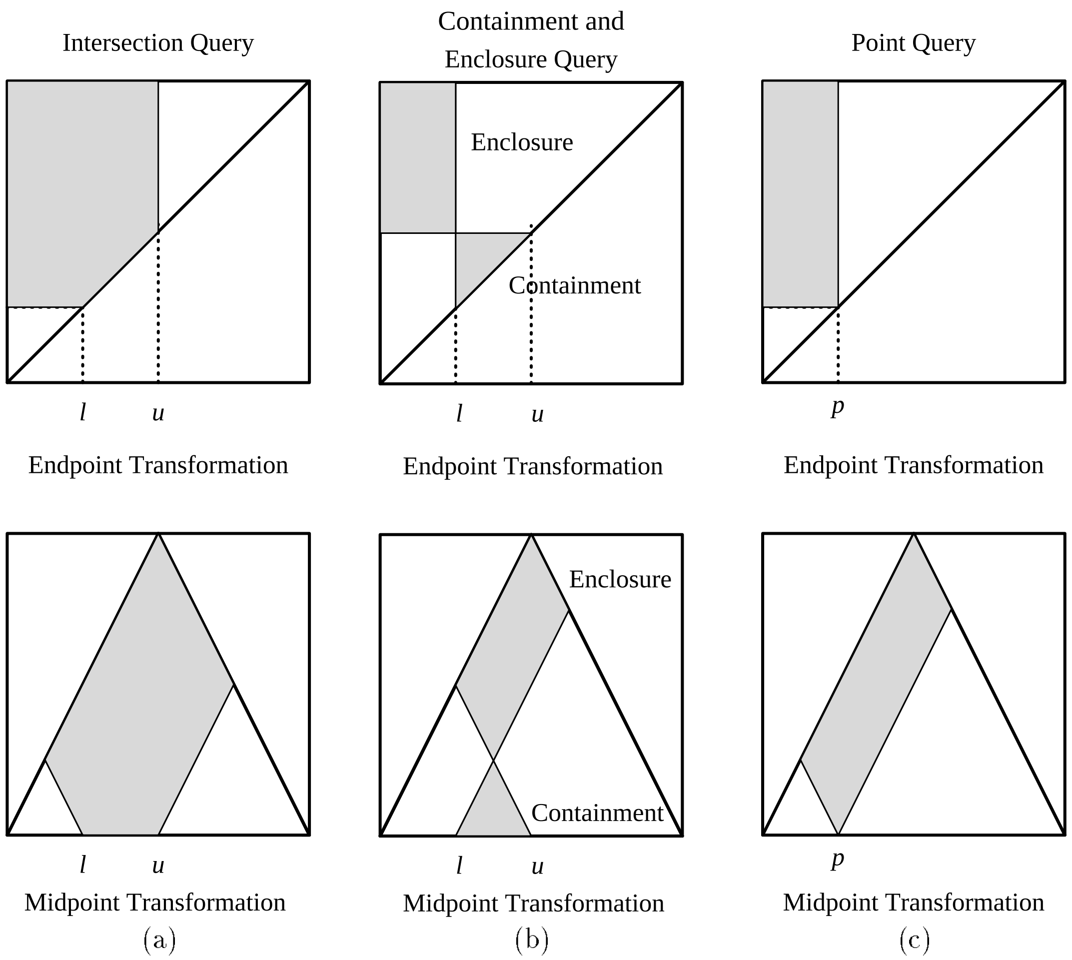

Figure 23: Search Queries in Dual Space

which may occur as a result of the transformation.

Finally, one has to keep in mind that the transformation approach is only applicable to "simple" objects, such as circles or rectangles. More complex entities, such as polygons with arbitrary shapes, must be approximated by an MBB or some other simple geometry before being transformed. In this case, using a PAM as described above only leads to a partial solution (cf. Fig. 2).

Space-filling curves (Sagan 1994) are an interesting alternative to the usual MBB approach that seems to suffer less from these drawbacks. In *z-ordering* (Orenstein and Merrett 1984), for example, a complex object is approximated not only by *one* simpler geometry, but by the union of *several* rectangular regions. Each region is transformed into a bitstring and stored in a B$^+$-tree. Section 5.2 contains further details.

5.1.2 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