Indexing in Spatial Databases

Beng Chin Ooi†
Dept. of Inf. Sys. and Comp. Sci.
National U. of Singapore
Kent Ridge Singapore 0511

Ron Sacks-Davis
Collaborative Inf. Tech. Res. Inst.
R.M.I.T. and The U. of Melbourne
Victoria Australia 3053

Jiawei Han‡
School of Computing
Simon Fraser U.
B.C. Canada V5A 1S6

Abstract

Spatial information processing has been a focus of research in the past decade. In spatial databases, data are associated with spatial coordinates and extents, and are retrieved based on spatial proximity. A formidable number of spatial indexes have been proposed to facilitate spatial data retrieval. In this paper, we examine various spatial indexes proposed in the literature and present a taxonomy on them. Each class of indexing structures is reviewed in two steps: first, the index structure is described; second, the strengths and weaknesses are identified. The paper may be used as a guideline in designing new indexing structures and serves as a basis in determining the suitability of a particular class of indexes for specific applications.

† The work of this author was supported in part by the National University of Singapore Research Grant NUS-910694.

‡ The work of this author was supported in part by the Natural Sciences and Engineering Research Council of Canada under the research grant OPG-3723.

1. Introduction

The typical notion of a database is that of a system in which one stores data about a fixed enterprise that one has modelled in some way. Quite often, the modelling is constrained in ways that help to reduce the complexity to a manageable level. The usual kinds of uses to which that data is put includes fairly straightforward applications like inventory management, shipping systems, and payroll systems. However, research in databases has advanced the state-of-the-art sufficiently that we are now moving into non-traditional applications that could not have been foreseen even ten years ago. This move has been largely demand-driven. For example, the emergence of spatial databases was initially driven by the demand for managing large volume of spatial data used in the geosciences and computer-aided design (CAD). The thrust was made more vigorous by newfound applications in computer vision and robotics, computer visualization, geographical information processing, automated mapping and facilities management. The kind of data modelling required is far more ambitious and complex than that required in the more traditional application areas.

But what is *spatial data*? Generally speaking, it is data that has, as a property, some connection with coordinates in a 2-dimensional, 3-dimensional space or even a higher dimensional space. Some examples of these are solids in CAD, components on printed-circuit boards, and roads and houses on maps. These objects can be broadly divided into three classes, namely, points, lines, polygons and volumetric objects. A *spatial database system* (SDS) is a general purpose software system that manages the storage and retrieval of data, ensures the consistency of data, and provides tools to reference spatial data by their positions and extensions. A spatial information system is a more general term to describe an SDS that has application customized tools for analyzing spatial properties of the data. It has to model reality, integrate spatial data acquired from different sources and by different means, and support processing and analysis of data on demand.

Spatial data are large in quantity and are complex in structures and relationships. Take geographic information systems, a popular type of spatial database systems, as an example. In such a system, the database is a collection of data objects over a particular multi-dimensional space. One such example is shown in Figure 1. The spatial description of objects is typically extensive, ranging from a few hundred bytes in land information system (commonly known as LIS) applications to megabytes in natural resource applications. As well, many spatial databases of real-world interest are very large, with sizes ranging from tens of thousands to millions objects. These spatial objects may *intersect*, be *adjacent* to others, or *contain* other objects. For the example shown in Figure 1, shop $s1$ is adjacent $s2$, and the park contains a children playground. Queries such as "display all the parks that contain a children playground" which involve the use of spatial operators are common. Spatial operators are much more expensive to

Figure 1 An example of spatial data

compute compared to the more conventional operators such as relational *join* and *select*. The efficiency of these operations is dictated by how the data is represented, and how fast the relevant data for a particular computation can be retrieved. The representation scheme which maps the original data objects into a set of objects in the storage medium is crucial to the computation of the operators. The sheer volume of the data being retrieved means that one must minimize the number of disk pages being accessed and the amount of redundant data being fetched and manipulated.

Efficient processing of queries manipulating spatial relationships relies upon auxiliary indexing structures. Due to the volume of the set of spatial data objects, it is highly inefficient to precompute and store spatial relationships among all the data objects (although there are some proposals that store precomputed spatial relationships [LuH92, Rot91]). Instead, spatial relationships are materialized dynamically during query processing. In order to find spatial objects efficiently based on proximity, it is essential to have an index over spatial locations. The underlying data structure must support efficient spatial operations, such as locating the neighbors of an object and identifying objects in a defined query region.

Most indexes are based on the principle of divide and conquer [AHU74]. Indexing structures following this approach are typically hierarchical. The approach is naturally suitable for a database system where the memory space is limited, and hence the pruning of a search must be performed such that the more detail to be examined, the smaller number of objects are being examined. An advantage of hierarchical structures is that they are efficient in range searching. Indexing in a spatial database (SD) is different from indexing in a conventional database in that data in an SDS are multi-dimensional objects and are associated with spatial coordinates. The search is based not on the attribute values but on the spatial properties of objects.

Many spatial indexes in one way or another are based on some existing point indexes such as kd-trees [Ben75] and B^+-trees [Com79]. The techniques used to extend point indexes to accommodate spatial objects can be generally categorized into three classes [SeK88]: object mapping, object duplication and object bounding. The object mapping approach maps objects defined by n-vertices from a k-dimensional space into points in either a nk-dimensional space or single-value objects in the original k-dimensional space. The object duplication approach stores an object identifier in multiple data spaces that the object overlaps, while the object bounding approach applies hierarchical grouping to partition data objects into different groups with each representing a data space. Each approach has its own strength and weakness, which directly affects the performance of indexes adopting it.

In this paper, the spatial data structures proposed in the literature are analyzed. The work is focused on the indexing structures designed for non-zero sized objects. The review of these indexes is organized in two steps: first, the structures are described; second, their strength and weakness are highlighted. This paper concentrates on a spectrum of indexes suitable for spatial objects and provide a taxonomy of spatial indexing techniques. Due to the breadth of the topic, it is not our intention to omit any proposed indexing structures.

The paper is organized as follows. In Section 2, we briefly discuss various issues related to spatial processing. In Section 3, we describe the techniques used in extending point indexing structures for non-zero sized objects. A taxonomy of the indexing structures is also presented. Spatial indexing structures are presented in Section 4. Spatial indexes are categorized into subsections based on the base structures. In Section 5, issues on evaluating the performance of spatial indexes are discussed, and approaches adopted in the literature are reviewed. Finally, the study is summarized in Section 6.

2. Spatial Database Retrieval

Spatial databases generally refer to the collection of data which have spatial coordinates and are defined within a space. Spatial database systems are the software systems that support the the manipulation, storage and analysis of spatial data and display of data in visual form.

In SDS, spatial operators such as *intersect* and *contain* are supported. These operators are much more expensive to compute compared to the conventional join or selection operator. Conventional data models are not particularly suitable for geographic applications because they do not efficiently support the types of operations that are required for spatial applications and, they are not suitable for the storage and manipulation of spatial data and graphical data. In general, a SDS is different from a conventional DBMS in the following ways.

(1) The amount of data is usually very large (eg. contour information in GIS) and the graphical data is expensive to capture.

(2) Spatial data types consist of complex objects, such as lines and polygons.

(3) Spatial operators are often more complex than numeric operators.

(4) It is difficult to define spatial ordering for spatial objects.

Objects in SDS are of irregular shape, and the irregular shape of these objects is the main cause for expensive spatial operator computation. Consider one of the simplest operators, such as *intersection*. The intersection of two polyhedra requires testing all points of one polyhedron against the other. The intersection of two polyhedra objects is not always a polyhedron; the intersection may sometimes consist of a set of polyhedra. The efficiency of these operations depends on the representation of the data [Gun88, Sam89], storage of objects [LOD91, DrS93] and retrieval of relevant data for computation. When data objects are stored in disks, the semantics of the data must be captured so that they can be reconstructed correctly and efficiently. Representation schemes developed for geometric modeling [Man88, Gun88, Req80, Sam89] are suitable for spatial objects.

Objects in SDS, roads and lakes in GIS, do not conform to any fixed shape. An indexing structure which provides fast access to a particular set of data objects only helps reduce the number of pages being accessed and the number of redundant data being fetched and tested. It is expensive to perform any spatial testing (eg. intersection and containment) on their exact location and extent; therefore, some initial approximation or filtering is used. By far, the most commonly used approximation is the container approach. In the container approach, the minimum bounding rectangle/circle (box/sphere) — the smallest rectangle/circle (box/sphere) that encloses the object is used to represent an object, and only when the test on container succeeds then the real object is examined. The bounding box (rectangle) is used throughout in this paper as the approximation technique for discussion purposes. Bounding boxes allow efficient proximity query processing by preserving the spatial identification and dynamically eliminating many potential intersection tests efficiently. The fact that two objects are disjoint if their bounding boxes are reduces the testing cost since the test on the intersection of two polygons or a polygon and a sequence of line segments is much more expensive than the test on the intersection of two rectangles. The k-dimensional bounding boxes can be easily defined as a single dimensional array of k entries: $(I_0, I_1, ..., I_{k-1})$ where I_i is a closed bounded interval $[a, b]$ describing the extent of the spatial object along dimension i. Alternatively, the bounding box of an object can be represented by its centroid and extensions on each of the *k* directions.

The object approximation and spatial indexes supporting such concepts are used to eliminate objects that could not possibly contribute to the answer of queries. The search acts as a filter to reduce redundant objects, whose result is a set of candidates that includes all the answer and possibly some false hits. Each candidate is then examined to remove false hits from the set. The number of false hits is dependent on the approximation techniques. While the indexes aim to reduce the number of pages being accessed, the approximation techniques aim to reduce computation time. Objects extended diagonally may be badly approximated by bounding boxes, and false matches may result. If the approximation technique is very inefficient, yielding very rough approximations, additional page accesses will be incurred. More effective approximation methods include *convex hull* [PrS85] and *minimum bounding m-corner*. The covering polygons produced by these two methods are however not axis-parallel and hence incur more expensive testing. The construction cost of approximations and storage requirement are higher too.

Decomposition of regions into convex cells was proposed in [Gun88] to improve object approximation. Likewise, an object may be approximated by a set of smaller rectangles/boxes. In [AbS84], based the quad-tree tessellation approach, an object is decomposed into multiple sub-objects based on the quad-tree quadrants that contain them. The decomposition has its problem of having to store object identity in multiple locations in an index. The problems of the redundancy of object identifiers and the cost of object-reconstruction can be very severe if the decomposition process is not carefully controlled. They can be controlled to a certain extent by limiting the number of elements generated or by limiting the accuracy of the decomposition [Ore90].

Approximations are widely recognized as an effective strategy in reducing the number of actual spatial testings [BKS93a]. Basic criteria in selecting an appropriate filtering technique include its approximation effectiveness, storage overhead, cost of testing, and cost of construction. One point to note here is that the efficiency of a dynamic index is independent from the choice of a flitering technique.

Spatial indexes are built based on the supported approximations. The operations offered by such an index are insert, delete and search. The commonly used conventional key-based range (associative) search, which retrieves all the data falling within the range of two specified values, is generalized to an *intersection search*. That is, given a query region, the search finds all objects that intersect it. The intersection search can be easily used to implement *point search* and *containment search*. For point search, the query region is a point, and its used to find all objects that contain it. Containment search is a search for all objects that are strictly contained in a given query region and it can be implemented by ignoring objects that fails such a condition in intersection search.

The search operation supported by an index can be used in facilitating a spatial selection or spatial join operation. While a spatial selection retrieves all objects of the same entity based on a spatial predicate, a spatial join is an operation that relates objects are two different entities based on a spatial predicate. The following illustrate two examples:

[1] Spatial Selection: Find all CITIES in California. Assume that the spatial description of California is known, it is used as a search region to find all cities that are contained in it.

[2] Spatial Join: Find all PARKs that are adjacent to SHOPPING-COMPLEX. Each object of a spatial entity, say SHOPPING-COMPLEX, is used as a search region to find all adjacent PARK objects whose approximations intersect.

For spatial selections, the whole relation has to be searched if no spatial index has been supported. The join operation is even more expensive, as for each object of a relation, the second relation has to be entirely accessed. With the use of indexes, only a small portion of the relation has to be searched. The operation of a spatial join can be further improved by simultaneous traversal of the indexes of two join relations [BKS93]. This is possible since spatial indexes hierarchically partition the space into subspaces, and the information about space is captured in internal nodes. Take intersection for example, two objects will definitely not intersect if the space that contain each of them do not. Based on such fact, only spaces that may contain intersecting objects are searched.

3. The Taxonomy

Non-spatial data are different from spatial ones in that spatial data are associated with spatial locations, and many-to-many spatial relationships exist among spatial objects. It is possible to represent some frequently referenced relationships using specially constructed relations or fragments of existing relations, or by maintaining conventional links. However, it is not pragmatic to pre-materialize all such relationships. Consequently, dynamic evaluation of spatial relationships is necessary.

The basic concept of indexing spatial data in a space is to partition the space into a manageable number of smaller subspaces, which are in turn further partitioned into even smaller subspaces, and so on. The process repeats until each unpartitioned subspace contains a small number of objects which can be stored in a data page. Different space partitioning strategies may be designed to reduce retrieval time and storage space. By doing so, a hierarchy of spaces is formed, which should be organized using an appropriate data structure to reduce retrieval time. Similar to other databases, the two basic issues that need to be addressed in designing an indexing structure for an SDS are: *the efficient use of storage* and *the ease of information retrieval*.

Data structures of various types, such as B-trees [BaM72, Com79], ISAM indexes, hashing and binary trees [Knu73], have been used as a means for efficient access, insertion and deletion of data in large databases. All these techniques are designed for indexing data based on primary keys. To use them for indexing data based on secondary keys, inverted indexes are formed. However, this technique is not adequate for a database where range searching on secondary keys is a common operation. For this type of applications, multi-dimensional structures, such as grid-files [NHS84], multi-dimensional B-trees [Kri84, OuS81, ScO82], kd-trees [Ben75] and quad-trees [FiB74] were proposed to index multi-attribute data. Such indexing structures are known as point indexing structures as they are designed to index data objects which are points in a multi-dimensional space.

Spatial search is similar to non-spatial multi-key search in that coordinates may be mapped onto key attributes and the key values of each object represent a point in a *k*-dimensional space. However, spatial objects often cover irregular areas in multi-dimensional spaces and thus cannot be solely represented by point locations. Although techniques such as mapping regular regions to points in higher dimensional spaces enable point indexing structures to index regions, such representations do not help support spatial operators such as intersection and containment.

Based on the classification techniques proposed in [Lom92, SeK88], the techniques used for adapting existing indexes into spatial indexes can be generally classified as follows:

(1) *The Transformation Approach*:

Parameter Space Indexing: Objects with n vertices in a k-dimensional space are mapped into points in an nk-dimensional space. For example, a two-dimensional rectangle described by the bottom left corner (x_1, y_1) and the rop right corner (x_2, y_2) is represented as a point in a four-dimensional space, where each attribute is

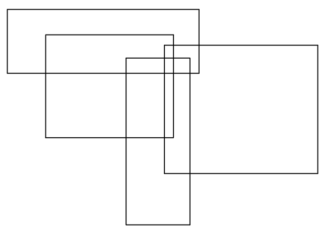


Figure 2 Highly dense data

taken as from a different dimension. After the transformation, points can be stored directly in existing point indexes. An advantage of such an approach is that there is no major alteration of the multi-dimensional base structure. The problem with the mapping scheme is that the *k*-dimensional objects that are spatially close in a *k*-dimensional space may be far apart when they are represented as points in an *nk*-dimensional space. As a consequence, intersection search can be inefficient. Also, the complexity of insertion operation typically increases with higher dimensionality.

Mapping to Single Attribute Space: The data space is partitioned into grid cells of the same size, which are then numbered according to some curve-filling methods. A spatial object is then represented by a set of numbers or one-dimensional objects. These one-dimensional objects can be indexed using conventional indexes such as B^+-trees. A B^+-tree can be used directly to index objects that have been mapped from a *k*-dimensional space into points in a one-dimensional space.

(2) *The Non-Overlapping Native Space Indexing Approach*

Object Duplication: A *k*-dimensional data space is partitioned into pairwise disjoint subspaces. These subspaces are then indexed. An object identifier is duplicated and stored in all of the subspaces it intersects; that is, an object identifier may be stored in multiple pages.

Object Clipping: A technique similar to object duplication object clipping. Instead of duplicating the identifier, an object is decomposed into several disjoint smaller objects so that each smaller sub-object is totally included in a subspace.

The most important property of object duplication or clipping is that the data structures used are straightforward extensions of the underlying point indexing structures. Also, both points and multi-dimensional non-zero sized objects can be stored together in one file without having to modify the structure. However, an obvious drawback is the duplication of objects which requires extra storage and hence more expensive insertion and deletion procedures. Another limitation is that the density (i.e., the number of objects that contain a point) in a map space must be less than the page capacity (i.e., the maximum number of objects that can be stored in a page). Figure 3 illustrates this point; no matter where a split is introduced, there will be four rectangles in one of the resulting areas.

(3) *The Overlapping Native Space Indexing Approach*

The basic idea to indexing spatial database is to hierarchically partition its data space into a manageable number of smaller subspaces. While a point object is totally included in an unpartitioned subspace, a non-zero sized object may extend over more than one subspace. Rather than supporting disjoint subspaces as in the non-overlapping space indexing approach, the overlapping native space indexing approach allows overlapping subspaces such that objects are totally included in only one of the subspaces. These subspaces are organized as a hierarchical index and spatial objects are indexed in their native space.

A major design criterion for indexes using such an approach is the minimization of both the overlap between bounding subspaces and the coverage of subspaces. A poorly designed partitioning strategy may lead to unnecessary traversal of multiple paths. Further, dynamic maintenance of effective bounding subspaces incurs high overhead during updates.

A number of indexing structures use more than one extending technique. Since each extending method has its own weaknesses, the combination of two or more methods may help to compensate the weaknesses of each other. However, an often overlooked fact is that the use of more than one extending method may also produce a counter effect: inheriting the weaknesses from each method.

Figure 4 shows the evolution of the spatial indexing structures. A solid arrow indicates a relationship between a new structure and the original structures that it is based upon. A dashed arrow indicates a relationship between a new structure and the structures from which the techniques used in the new structure originated, even though some were proposed independent of the others. In the diagram and also in the next section, the indexes are classified into four groups based on their base structures: namely, binary trees, B-trees, hashing, and space filling methods.

Most spatial indexing structures (e.g. R-trees, R^*-trees, skd-trees) are nondeterministic in that different sequences of insertions result in different tree structures and hence different performance even though they have the same set of data. The insertion algorithm must be dynamic so that the performance of an index will not be dependent on the sequence of data insertion. During the design of a spatial index, issues that need to be minimized are:

(a) The area of covering rectangles maintained in internal nodes;

(b) The overlaps between covering rectangles for indexes developed based on the overlapping native space indexing approach;

(c) The number of objects being duplicated for indexes developed based on the non-overlapping native space indexing approach;

(d) The directory size, and its height.

There is no straight-forward solution to fulfill all the above conditions. The fulfillment of the above conditions by an index can generally ensure its efficiency, but this may not be true for all the applications. The design of an index needs to take the computation complexity into consideration as well, which although is a less dominant factor considering the increasing computation power of today’s systems. Other factors that

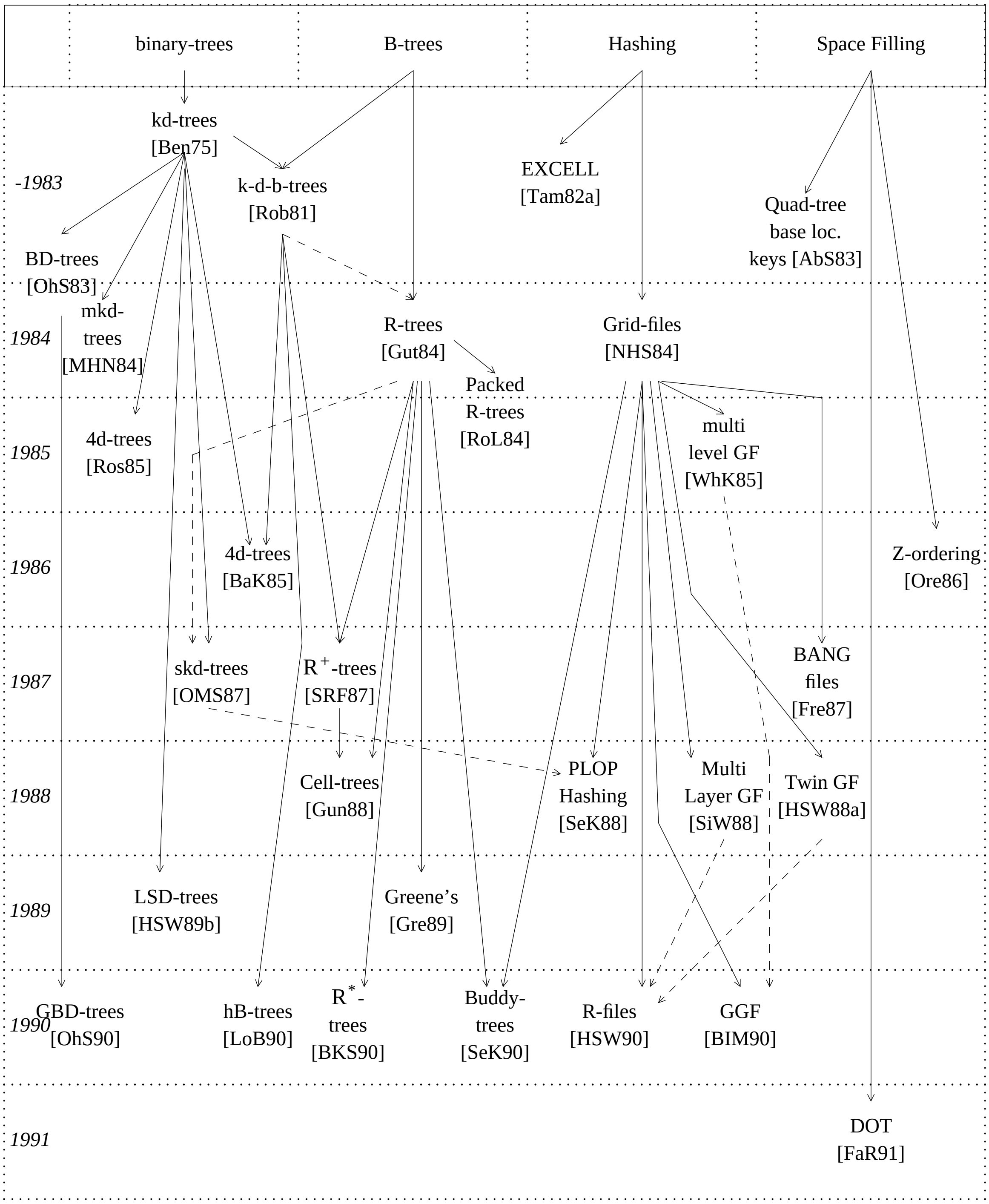


Figure 4 The evolution of the spatial indexes

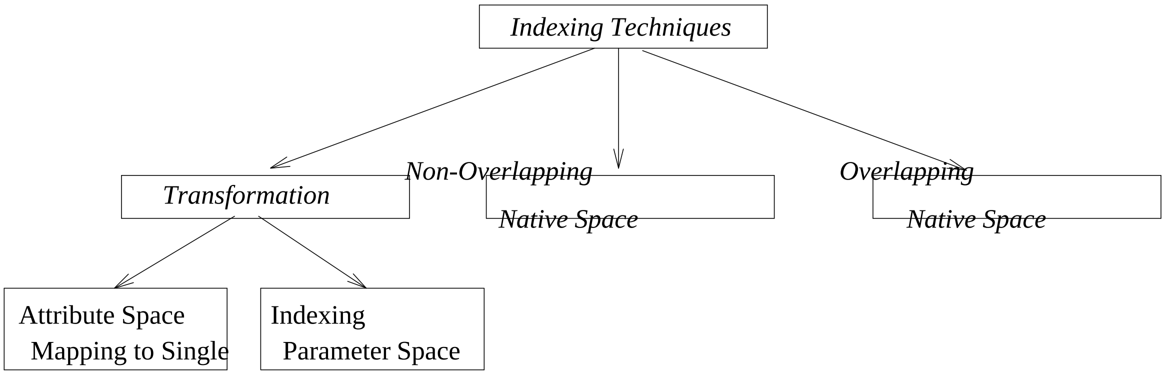


Figure 4 Classification of indexing structures

affect the performance of information retrieval as a whole include buffer design, buffer replacement strategies, space allocation on disks, and concurrency control methods.

4. Access Methods for Extended Spatial Objects

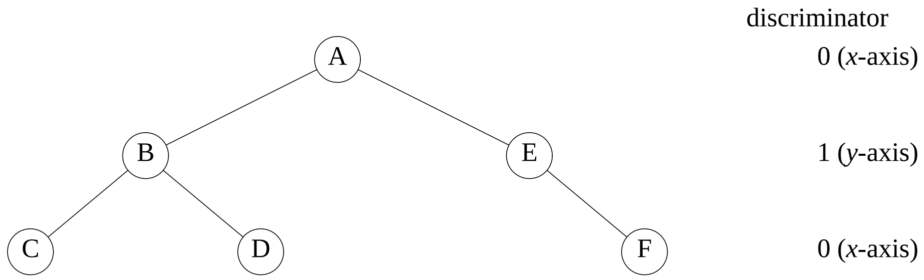
In this section, we review indexes based on its basic structure and operations, and efficiency.

4.1. Binary-Tree based Indexing Techniques

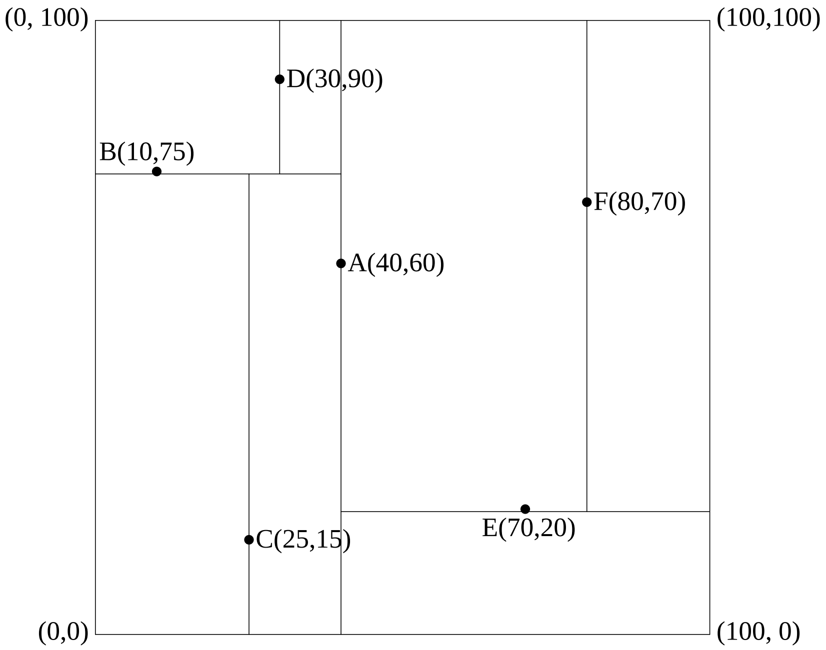
The binary search tree is a basic data structure for representing data items whose index values are ordered by some linear order. The idea of repetitively partitioning a data space has been adopted and generalized in many sophisticated indexes. In this section, we will examine indexes originated from the basic structure and concept of binary search trees.

4.1.1. The kd-Tree

The *kd-tree* [Ben75], a k-dimensional binary search tree, was proposed by Bentley to index multi-attribute data. A node in the tree (Figures 5a and 5b) serves two purposes: representation of an actual data point and direction of a search. A discriminator whose value is between 0 and *k*-1 inclusive, is used to indicate the key on which the branching decision depends. A node *P* has two children, a left son *LOSON(P)* and a right son *HISON(P)*. If the discriminator value of node *P* is the *jth* attribute (key), then the *jth* attribute of any node in the *LOSON(P)* is less than *jth* attribute of node *P*, and the *jth* attribute of any node in the *HISON(P)* is greater than or equal to that of node *P*. This property enables the range along each dimension to be defined during a tree traversal such that the ranges are smaller in the lower levels of the tree.



(a) The structure of a kd-tree



(b) The planar representation

Figure 5 The organization of data in a kd-tree

Complications arise when an internal node is deleted. When an internal node is deleted, say *Q*, one of the nodes in the subtree whose root is *Q* must be obtained to replace *Q*. Suppose *i* is the discriminator of node *Q*, then the replacement must be either a node in the right subtree with the smallest *ith* attribute value in that subtree, or a node in the left subtree with the biggest *ith* attribute value. The replacement of a node may also cause successive replacements.

To reduce the cost of deletion, a non-homogeneous kd-tree [Ben79a] was proposed. Unlike a homogeneous index, a non-homogeneous index does not store data in the internal nodes and its internal nodes are used merely as directory. When splitting an internal node, instead of selecting a data point, the non-homogeneous kd-trees selects an arbitrary hyperplane (a line for the two dimensional space) to partition the data points into two groups having almost the same number of data points and all data points resides in the leaf nodes.

The kd-tree has been the subject of intensive research [BaK86, BER85a, BER85b, BER85c, BeF79, Ben79b, ChF79, EaZ82, FBF77, LeW77, MHN84, OhS83, Ore82, OvL82, Rob81, Ros85, ShB78, ShR85, etc.] over the past decade. Many variants have been proposed in the literature to improve the performance of the kd-tree with respect to issues such as clustering, searching, storage efficiency and balancing.

4.1.2. Kd-tree Extensions with Paging Capability

An indexing tree is often too large to be stored in main memory; it has to be paged into disks. Kd-Trees can be stored in disk by using binary search tree paging techniques [CeS82] or tree organization of B-trees [BaM72].

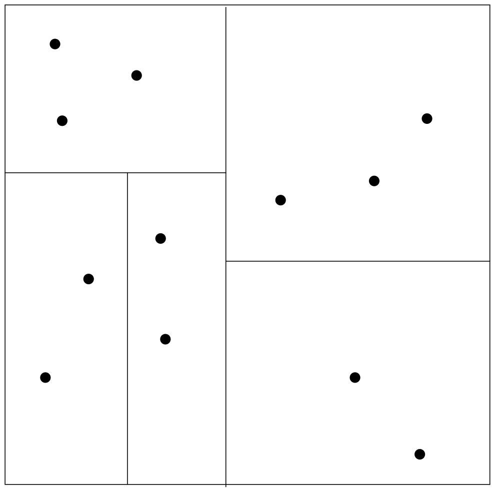
The K-D-B-Tree

To improve the paging capability of the kd-tree, the K-D-B-Tree [Rob81] which is a combination of a kd-tree and a B-tree [BaM72, Com79] was proposed.

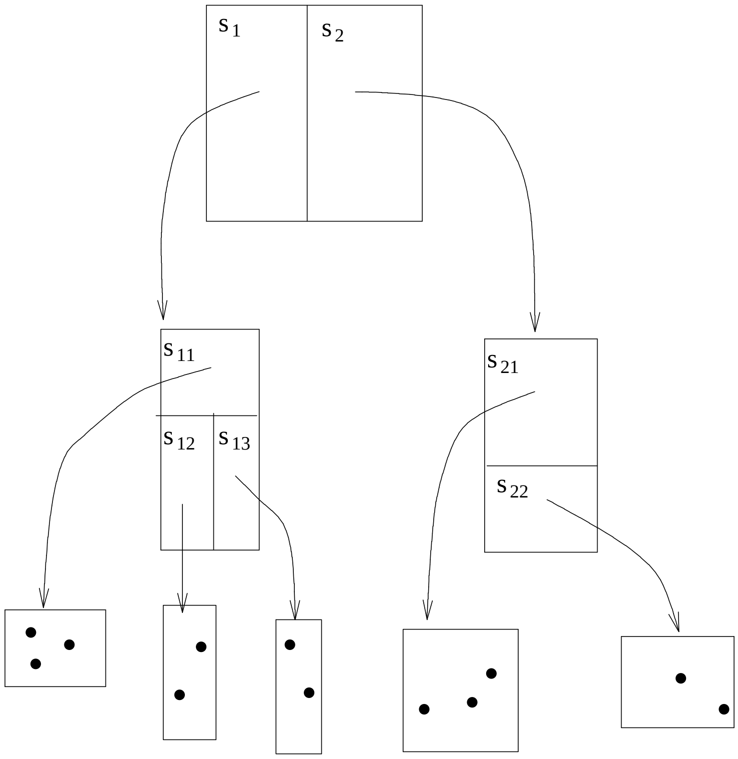
The K-D-B-tree consists of two basic structures: region pages and point pages (see Figure 6). While point pages contain object identifiers, region pages store the descriptions of subspaces in which the data points are stored and the pointers to descendant pages. Note that in a non-homogeneous kd-tree [Ben79a], a space is associated with each node: a global space for the root node, and an unpartitioned subspace for each leaf node. In the K-D-B-tree, these subspaces are explicitly stored in a region page. These subspaces (e.g. s_{11}, s_{12} and s_{13}) are pairwise disjoint and together they span the rectangular subspace of the current region page (eg. s_1), a subspace in the parent region page.

During insertion of a new point into a full point page, a split will occur. The point page is split such that the two resultant point pages will contain almost the same number of data points. Note that a split of a point page requires an extra entry for the new point page, this entry will be inserted into the parent region page. Therefore, the split of a point page may cause the parent region page to split as well, which may further ripple all the way to the root; thus the tree is always perfectly height-balanced.

When a region page is split, the entries are partitioned into two groups such that both have almost the same number of entries. A hyperplane is used to split the space of a



(a) Planar partition



(b) A hierarchical K-D-B-tree structure

Figure 6 A K-D-B-tree structure

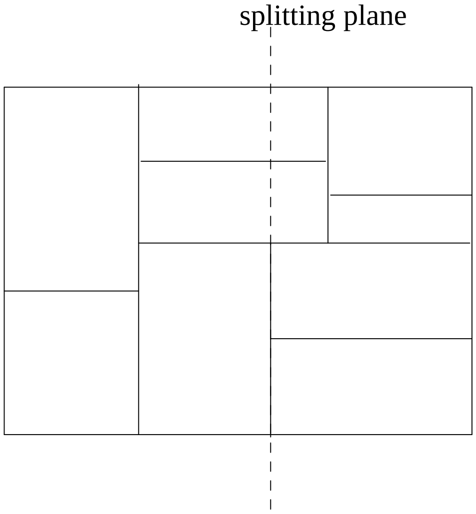


Figure 7 The splitting of a node causing further splits

region page into two subspaces and this hyperplane may cut across the subspaces of some entries. Consequently, the subspaces that intersect with the splitting hyperplane must also be split so that the new subspaces are totally contained in the resultant region pages. Therefore, the split may propagate downward as well. An example is shown in Figure 7, in which two child pages must be split. If the constraint of splitting a region page into two region pages containing about the same number of entries is not enforced, then downward propagation of split may be avoided. The choice of the dimension for splitting and the splitting point would be chosen so that both resultant pages have almost the same number of entries and the number of splittings is minimized. However, no details on selection of splitting points are given in [Rob81].

The upward propagation of a split will not cause the underflow of pages but the downward propagation is detrimental to storage efficiency because a page may contain less than the usual page threshold, typically half of the page capacity. To avoid unacceptably low storage utilization, local reorganization can be performed. For example, two or more pages whose data space forms a rectangular space and who have the same parent can be merged followed by a resplit if the resultant page overflows. In Figure 7, the reorganization of s_{12} (s_{13}) involves merging (and splitting) with only s_{13} (s_{12}), whereas region of s_{11} requires merging with both s_{12} and s_{13}.

The K-D-B-tree has incorporated the pagination of the B-tree and the tree is height-balanced as a result. Nevertheless, poorer storage efficiency is the trade-off.

The hB-Tree

In the K-D-B-tree, a region node is split by cutting the region with a plane, posibly cutting through some subregions as well. The child nodes with their space being cut must also invoke the splitting process, causing sparse nodes at lower levels. To overcome such a problem, a new multi-attribute index structure called the holey brick B-tree (the hB-tree) [LoS89, LoS90] allows the data space to be holey, enabling removal of any data subspace from a data space. The concept of holey bricks has been used in [OhS83] as an attempt to improve the clustering of data in a kd-tree known as the BD-tree. The hB-tree structure is based on the K-D-B-tree structure, but it allows the data space associated with a node to be non-rectangular and it uses kd-trees for space representation in its internal nodes. The hB-tree is a height-balanced tree. In an hB-tree, the leaf nodes are known as *data nodes* and the internal node as *index nodes*. An index node data space is a union of its child node subspaces which are obtained through kd-tree recursive partitioning.

A k-dimensional data space represented by its boundaries requires *2k* coordinates. To obtain a data space of interest to the search, half of the data subspaces in a node have to be searched on average and for each data space, $2k$ comparisons are required. For m data spaces, we need on average $m*k$ comparisons.

The m data subspaces derived through kd-tree recursive partitioning can be represented by a kd-tree with m–1 kd-tree nodes. It requires one comparison at each internal and $2k$ for comparison with the unpartitioned subspace. The average number of comparisons is much smaller than that of the boundary representation. The use of kd-trees therefore reduces the search time as well as the storage space requirement.

(a) The internal structure of an hB-tree index node

(b) The resultant pages after a split

Figure 8 The hB-Tree

Like conventional kd-trees, internal nodes of the kd-tree structure in an hB-tree index node partition the search space recursively. Its leaf nodes reference some index nodes of the hB-tree. However, multiple leaves of a kd-tree structure may refer to the same hB-tree index node (see Figure 8a), giving rise to the "holey brick" representation. As such, the hB-tree is not truly a tree. During a split, the kd-tree is split into two subtrees, with each having between $\frac{1}{3}$ and $\frac{2}{3}$ of the nodes. In order to achieve this, a subtree may have to be extracted from the original tree structure. This causes duplication of a portion of the tree close to the root in the parent index node. A leaf node of such a kd-tree references either an hB-tree data node, an index node, or a marker (*ext* in Figure 8b) indicating that a subtree has previously been extracted and is referenced from a higher level index node. An example of split is illustrated in Figure 8. The deletion algorithm is not addressed in the paper.

Matsuyama's kd-Tree

While most kd-trees are proposed as point access methods, the kd-tree proposed by Matsuyama et al. [MHN84] is designed for two-dimensional non-zero sized spatial objects by supporting duplications of objects. The directory is a kd-tree, and for each leaf node, a data page is associated. A data page contains the identifiers of objects which are partially or totally included in its data space. Objects that overlap multiple unpartitioned data space are duplicated in respective data pages.

Matsuyama's kd-tree is searched like a conventional kd-tree. However, to insert an object, the object identifier needs to be inserted into all the pages with subspaces that intersect with the data object. It is quite common that object identifiers may be duplicated in more than one page, particularly when the sizes of objects are large. Whenever a page overflows, the page is split with a partition being introduced along the longer side of the rectangle. The subspace is partitioned into two subspaces and the two new pages contain all objects that intersect with their subspace.

To delete an object, it is necessary to search for all leaf nodes with subspaces that intersect with the data object and delete all identifiers referring to the data object. If the deletion of an object causes a page to be empty, the corresponding leaf node is marked *NIL*. To simplify the deletion algorithm, the underflowed data pages are not merged.

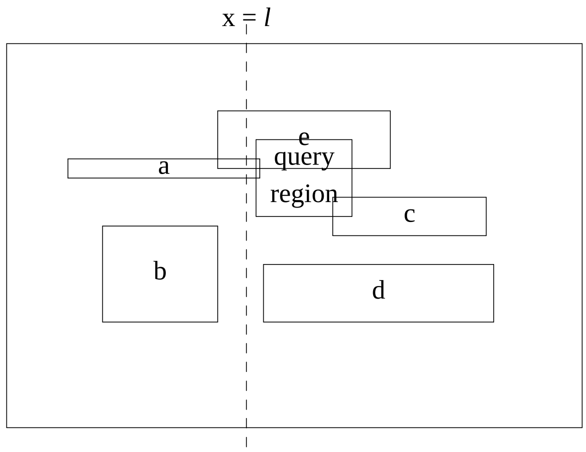
Matsuyama's kd-tree is one of the earlier indexing structures adopting the object duplication approach. Such an index is not suitable for indexing large objects as the overhead of redundant storage can be very high.

The 4-D-Tree

The kd-tree [Ben75] can be used to index two-dimensional rectangular objects by mapping the objects into points in a four-dimensional space [Ros85]. Each two-dimensional rectangle described by (*x1*, *y1*) and (*x2*, *y2*), is treated as a four attribute tuple (*x1*, *y1*, *x2*, *y2*). The discriminators are used cyclically and the nodes at the same level use the same discriminator. In [Ros85], the issues involved in mapping the data structure onto pages in secondary memory were not addressed. The same approach for the K-D-B-tree [Rob81] was suggested by Banerjee and Kim [BaK86]. The structure is known as the 4-d-tree.

The region search presented in [Ben75] can be used for a spatial intersection search. At each node of the 4d-tree, a discriminator (x_1, x_2, y_1 or y_2), discriminator value and pointers to two children nodes is stored. A two-dimensional subspace is associated with each node and as the tree is traversed during query, starting from the root, these subspaces are successively pruned. Let the query region be (qx_1, qx_2, qy_1, qy_2). Then, at each internal node, one of the conditions, $x_1 \leq qx_2$, $x_2 \geq qx_1$, $y_1 \leq qy_2$ or $y_2 \geq qy_1$, has to be used depending on the discriminator stored in that node in order to determine whether both subtrees or only one of the subtrees will need to be searched.

The important part in the search algorithm is the determination of the subspaces that bound the objects in the LO (left) and HI (right) subtrees. Traversal starts at the root with



disc = x_1

disc-value = *l*

objects stored in the LO subspace = {a, b, e}

objects stored in the HI subspace = {c, d}

Figure 9 A 4d-tree objects distribution the map as the associated space. Assume that the first discriminator is X_1, the LO subtree contains objects whose X_1 coordinate is less than the *discriminator value*, and the HI subtree contains objects whose X_1 coordinate is greater than the *discriminator value*. The X_1 values of the HI subspace are bounded below by the *discriminator value* and this fact can be used to reduce the subspace associated with the HI subspace. For example, to search for objects that overlap a given object with x_2 less than l (discriminator value) in Figure 9, we can conclude straight away that the right subtree does not contain any objects that will intersect with the given object. However, it is not possible to reduce the size of the LO subspace. Suppose the original map space is (x_1, x_2, y_1, y_2). Then the LO subspace is the same as that of the root node while the HI subspace is (*disc_value*, x_2, y_1, y_2). The problem is that the X_2 values of rectangles in the left subspace may fall on the right subspace, and there is no information about extent to which they overlap. At the next level, the HI subspace remains unchanged, but for the LO subspace X_2 is bounded by the current discriminator value. Hence, it is common that both subtrees of a node will need to be searched. Figure 9 illustrates the case where both subspaces have to be searched. The major problem associated with the 4-d-tree is its intersection search, which can be very costly due to the need for traversal of both subtrees when a query region lies in a subspace that cannot not bounded tightly using the discriminator values.

The Skd-Tree

Ooi et al [OMS87, OSM91] developed an indexing structure called the spatial kd-tree (the skd-tree) in an attempt to avoid object duplication and object mapping. At each node of a kd-tree, a value (the discriminator value) is chosen in one of the dimensions to partition a *k*-dimensional space into two subspaces. The two resultant subspaces, *HISON* and *LOSON*, normally have almost the same number of data objects. Point objects are totally included in one of the two resultant subspaces, but non-zero sized objects may extend over to the other subspace. To avoid the division of objects for and the duplication of identifiers in several subspaces, and yet to be able to retrieve all the wanted objects, [OSM87] introduced a virtual subspace for each original subspace such that all objects are totally included in one of the two virtual subspaces. With this method, the placement of an object in a subspace is based solely upon the value of its centroid.

One additional value for each subspace is stored: the maximum (max_{LOSON}) of the objects in the LOSON subspace, and the minimum (min_{HISON}) of the objects in the HISON subspace, along the dimension defined by the discriminator. The structure of an internal node of the skd-tree consists of two child pointers, a discriminator (0 to k–1 for a k-dimensional space), a discriminator-value, (max_{LOSON}) and (min_{HISON}) along the dimension specified by *discriminator*. The maximum range value of *LOSON* (max_{LOSON}) is the nearest virtual line that bounds the data objects whose centroids are in the *LOSON* subspace, and the minimum range value of *HISON* ($\min_{HISON}$) is the nearest virtual line that bounds the data objects whose centroids are in the *HISON* subspace.

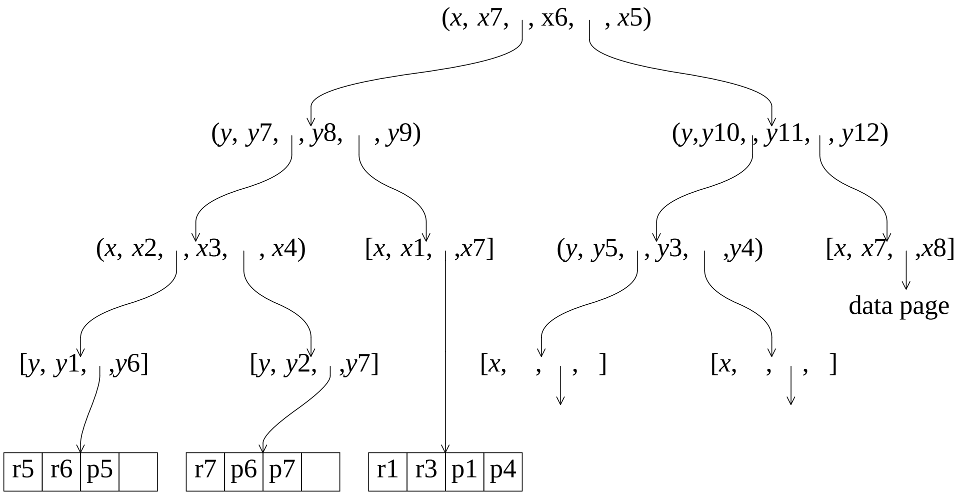
Leaf nodes contain *min-range* and *max-range* (in place of $\max_{LOSON}$ and $\min_{HISON}$ of an internal node) respectively, describing the minimum and maximum values of objects in the data page along the dimension specified by *bound*, and a pointer to the secondary page which contains the object bounding rectangles and identifiers. The minimum and maximum values could be kept for *k* dimensions. However, for storage efficiency, the range along one dimension that results in the smallest bounding rectangle is chosen. In [Ooi90], it was shown that such a range increases the height of the tree when it is stored as a multiway tree, and hence the improvement becomes fairly marginal.

Figures 10a and 10b show the structure of a two-dimensional skd-tree and illustrate the virtual boundary (dotted line), min_{HISON} or max_{LOSON} of each resultant subspace.

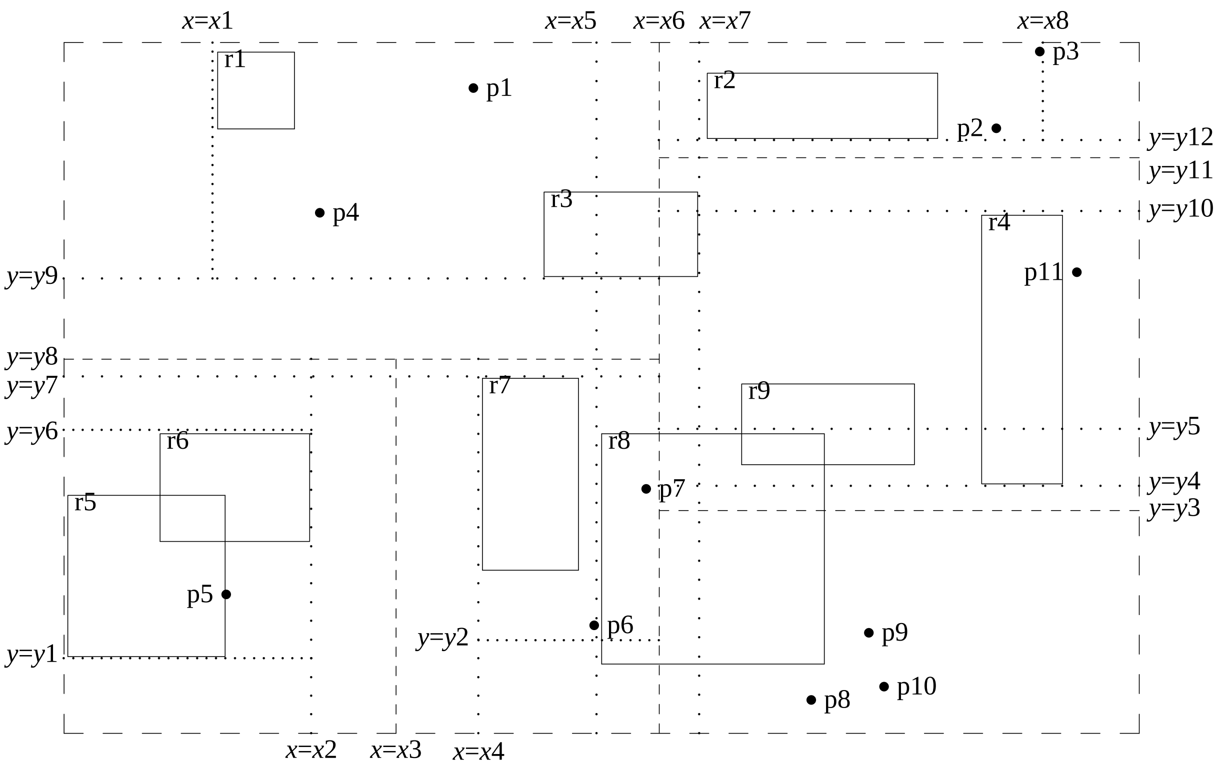
An implicit rectangular space is associated with each node and it is materialized during traversal. This rectangle is tested against the query region, and the subtree is examined if they intersect. Since the virtual boundary may sometimes bound the objects tighter than the partitioning line, the intersection search takes advantage of the existing virtual boundary to prune the search space efficiently. To further exploit the virtual boundaries, containment search which retrieves all spatial objects contained in a given query rectangle was proposed. During tree traversal, the algorithm always selects the boundaries that yield smaller search space. The direct support of containment search is useful to operators like *within* and *contain*. The search rapidly eliminates all objects that are not totally contained in the query region.

Inserting index records for new data objects is similar to insertion into a point kd-tree. As new index records are added to a bucket, the bucket is split if it overflows. At each node, the algorithm uses the centroid of the bounding rectangle of the new object to determine which subspace the object will be placed, and updates the virtual boundary if necessary.

To delete an object, the centroid of its bounding rectangle is used to determine where the object resides. The removal of an object may cause a bucket to underflow, and merging or reinsertion is then required. If the neighboring node is a leaf-node, then the two buckets are merged and the resultant bucket is resplit if overflow occurs. Otherwise, the records are required to be inserted into the neighboring subtree, and the neighboring node is promoted to replace the parent node. The merging follows the principle of buddy system as in [NHS84]; that is, the region of two merged nodes is rectangular and a proper subspace derivable from discriminator values in parent nodes. The major problem with deletion occurs when an object contributes to the boundary of a virtual space is deleted.



(a) A 2-d directory of the skd-tree



(b) A 2-d space coordinate representation

Figure 10 The structure of a spatial kd-tree

A new tighter boundary needs to replaces the old boundary which may not be as effective. The operation can be expensive as several pages whose space is adjacent to the deleted boundary need to be searched. The operation cost can be reduced by periodically sweeping the subtrees that are affected by deletions. It should be noted that the delay of finding replacements does not result in any invalid answer.

The directory of the skd-tree is stored in secondary memory. The bottom-up approach for binary tree paging proposed by [CeS82] is modified to store the skd-tree as a multiway-tree. When such a page splits, one of the subtrees is migrated to an existing page that can accommodate the subtree or a new page, and the root of the subtree is promoted to the parent page.

In [OSM91], it was shown that the containment search is insensitive to the different sizes of objects and distribution of objects, and it is always more efficient than the intersection search due to a smaller search space. It can be noticed that the leaf nodes of the skd-tree take up about half of the storage requirement for the directory. The main objective of having such layer of leaf nodes is to reduce the fetching of data pages. Experiments were conducted in [Ooi90] to evaluate the performance of skd-trees with and without the leaf nodes, under different data distributions. The experiments show that for uniform distributions of spatial objects, the leaf nodes do help reduce the page accesses. However, when the distributions are skewed, the extra layers are not effective and large directory sizes incur more page reads than that by the modified skd-tree. The modified skd-tree, which has less number of nodes, saves up to 40 % of the directory storage space.

The BD- and GBD-Trees

The BD-tree [OhS83], a variant of kd-trees, allows a more dynamic partitioning of space. Each non-leaf node in the BD-tree contains a variable-length string, called the *discriminator zone* (DZ) *expression*, consisting of 0's and 1's. The 0 means "<" and 1 "≥", with the leftmost digit corresponding to the first binary division, and the n-th bit corresponding to the n-th binary division. The string describes the left subspace while the right subspace is its complement (see example in Figure 11). Each string uniquely describes a space. A data space with the DZ expression (eg. 0100) which is the initial substring of a longer DZ expression (eg. 010001) encloses its data space. A BD-tree is different from a kd-tree in the following aspects. One, the data space of a BD-tree node is not a hyper-rectangle. The use of complement makes the space holey. Two, unlike the conventional kd-tree, the use of DZ expression enables rotation, achieving a greater degree of balancing. Three, the partition divides a space into two equal sized subspaces. Four, the discriminators are used cyclically so that each bit of a DZ expression can be correctly associated with a dimension.

The BD-tree is expanded to a balanced multi-way tree called the GBD-tree (generalized BD-tree) [OhS90]. In addition to a DZ expression, a bounding rectangle is used to describe a data space that bounds the objects whose centroids fall inside the region defined by the DZ expression. Centroids of objects are used to determine placement of objects in the correct bucket. While a DZ expression is used to determine the position in the tree structure where an entity is located based on its centroid, a bounding rectangle is used in intersection search.

In an internal node, each entry describes a data space obtained through binary decomposition. The union of these data spaces forms the data space of the node. While the data spaces described by the entries' DZ expressions do not overlap, their associated bounding rectangles overlap. During point search of an entity, an inclusion check of the DZ expression of the entity is performed against the DZ expression of a node. For the data space that includes the entity, its subtree is traversed. For the intersection search, the bounding rectangles stored in a node are used instead to select subtrees for traversal.

When a leaf node is overflowed, it is split into two. A recursive binary decomposition on alternative axis is performed on the overflowed data space until a subspace contains at least $2(M+1)/3$ entries. While this smaller space has a new DZ expression, the other subspace takes the DZ expression of the space before splitting. We call such a space a *complementary subspace*. A new entry is inserted into the parent node and the affected bounding rectangles are re-adjusted accordingly.

In an internal node splitting, the subspaces are checked in decreasing order of their sizes to find a data space that contains almost $(M+1)/2$ entries. A data space described by the DZ expression e_1 contains the data space described by the DZ expression e_2, if e_1 forms the initial substring of e_2. In the testing, all DZ expressions must be checked. The worst case is when a node is split into two nodes respectively having M entries and one entry. The DZ expression obtained is used as the DZ expression of a new node. The other new node, which re-uses the original node, is assigned with the DZ expression of the original space. When an entry is deleted, a node may be underflowed. Like B-trees, tree collapsing is required.

Conceptually, the GBD-tree is similar to the BANG file. The use of bounding rectangles can be applied to the BANG file. The GBD-tree has been shown in [OhS90] better efficiency than the R-tree in terms of tree construction time for a small set of data.

The LSD-tree

As an improvement to the fixed size space partitioning of the grid files, a binary tree, called the Local Split Decision tree (LSD-tree), that supports arbitrary split position was proposed in [HSW89b]. A split position can be chosen such that it is optimal with respect to the current cell. The directory of an LSD-tree is similar to that maintained by the kd-tree [Ben75]. Each node of the LSD-tree represents one split and stores the split dimension (cf: the discriminator of kd-trees) and position (cf: the discriminator value of kd-trees), and each leaf node points to a data bucket.

In an LSD-tree, the nodes in a directory T are divided into two directories: the internal directory and the external directory. The internal directory consists of a subtree that contains the root and is stored in main memory. The external directory consists of multiway-trees and is stored in secondary memory. In an external directory page, the subtree is organized as a heap. When a directory page is split, the root node of that directory page is inserted into the directory T and the left and right subtrees are stored in two distinct directory pages. The main objective of the paging algorithm [HSW89a] is to ensure that the heights of multiway-trees differ by at most one directory page. The proposed paging strategy is similar to that proposed in [CeS82], although the algorithm in [CeS82] makes no distinction between the external and internal directories. The major difference is that the internal directory is restructured such that the heights of multi-way trees in the external directory always differ by at most one page. To achieve this, the nodes that are close to the boundary that separates the internal and external directories must be moved around between these two directories. Note that the size of the internal directory depends on the allocated internal memory. Like kd-trees, rotation of the tree is not possible. If the data is very skewed, the property of the height differences of at most one cannot be upheld.

The deletion algorithm is not presented. We believe that the deletion of [CeS82] can be applied here. To index non-zero sized objects, objects are mapped from a *k*-dimensional into points in a *nk*-dimensional space. In principle, the LSD-tree is similar to the kd-tree [Ben75] and its paging strategy is similar to that used in [CeS82, OMS87].

4.2. B-tree based Indexing Techniques

B^+-trees have been widely used in data intensive systems to facilitate query retrieval. The wide acceptance of the B^+-tree is its height-balanced elegant characteristic, making it ideal for disk I/O where data transfer is in the unit of page. It has become an underlying structure for many new indexes. In this section, we discuss indexes based on the concept of the hierarchical structure of B^+-trees.

4.2.1. The R-Tree

The R-tree [Gut84] is a multi-dimensional generalization of the B-tree, that preserves height-balance. Like the B-tree, node splitting and merging are required for inserting and deleting objects. The R-tree has received a great deal of attention due to its well defined structure and the fact that it is one of the earliest proposed tree structures for non-zero sized spatial object indexing. Many papers have used the R-tree as a model to measure the performance of their structures.

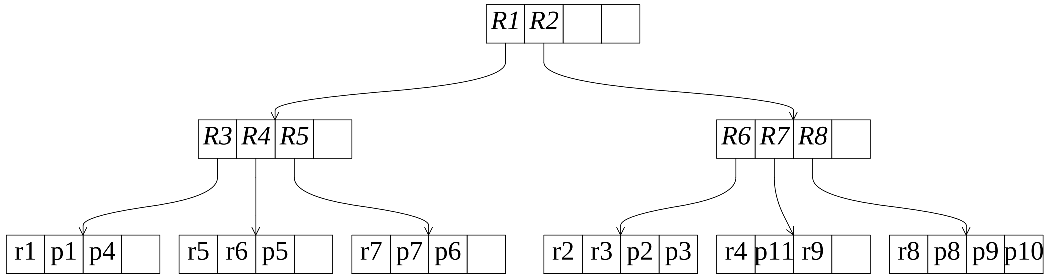
An entry in a leaf node consists of an *object-identifier* of the data object and a k-dimensional bounding rectangle which bounds its data objects. In a non-leaf node, an entry contains a *child-pointer* pointing to a lower level node in the R-tree and a bounding rectangle covering all the rectangles in the lower nodes in the subtree. Figures 12a and 12b illustrate the structure of an R-tree and its planar representation.

In order to locate all objects which intersect a query rectangle, the search algorithm descends the tree from the root. The algorithm recursively traverses down the subtrees of bounding rectangles that intersect the query rectangle. When a leaf node is reached, bounding rectangles are tested against the query rectangle and their objects are fetched for testing if they intersect the query rectangle.

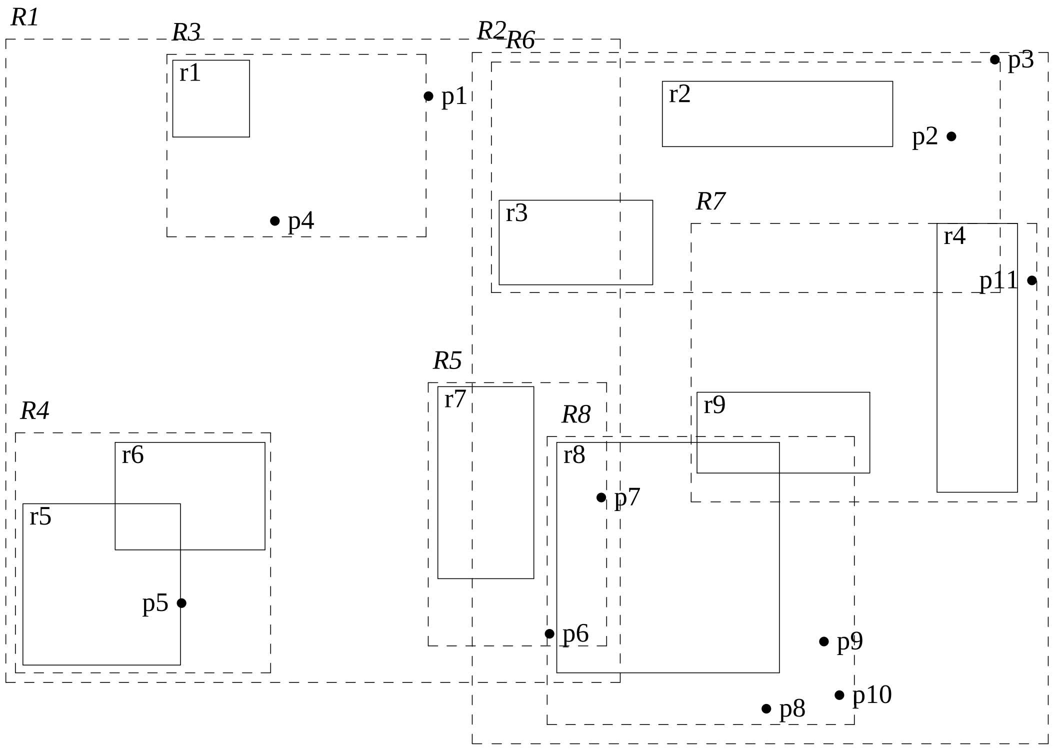
To insert an object, the tree is traversed and all the rectangles in the current non-leaf node are examined. The constraint of least coverage is employed to insert an object: the rectangle that needs least enlargement to enclose the new object is selected, the one with the smallest area is chosen if more than one rectangle meets the first criterion. The nodes in the subtree indexed by the selected entry are examined recursively. Once a leaf node is obtained, a straightforward insertion is made if the leaf node is not full. However, the leaf node needs splitting if it overflows after the insertion is made. For each node that is traversed, the covering rectangle in the parent is readjusted to tightly bound the entries in the node. For a newly split node, an entry with a covering rectangle that is large enough to cover all the entries in the new node is inserted in the parent node if there is room in the parent node. Otherwise, the parent node will be split and the process may propagate to the root.

To remove an object, the tree is traversed and each entry of a non-leaf node is checked to determine if the object overlaps its covering rectangle. For each such entry, the entries in the child node are examined recursively. The deletion of an object may cause the leaf node to underflow. In this case, the node needs to be deleted and all the remaining entries of that node are reinserted from the root. Similar to the node splitting, the deletion of an entry may cause further deletion of nodes in the upper levels. Thus, entries belonging to a deleted *ith* level node must be reinserted into the nodes in the *ith* level of the tree. Deletion of an object may change the bounding rectangle of entries in the ancestor nodes. Hence readjustment of these entries is required.

In searching, the decision whether to visit a subtree depends on whether the covering rectangle overlaps the query region. It is quite common for several covering rectangles in an internal node overlap the query rectangle, resulting in the traversal of several subtrees. Therefore, the minimization of overlaps of covering rectangles as well



(a) The directory of an R-tree



(b) A planar representation of an R-tree

Figure 12 The structure of an R-tree

as the coverage of these rectangles is of primary importance in constructing the R-tree [Gut84, RoL85].

The heuristic optimization criterion used in the R-tree is the minimization of the area of internal nodes covering rectangles. Two algorithms involved in the process of minimization are the insertion and its node splitting algorithms. Of the two, the splitting algorithm affects the index efficiency more. In [Gut84], splitting algorithms with exponential, quadratic and linear cost were discussed. It was shown that the performance of the quadratic and linear algorithms were comparatively similar. The quadratic algorithm in a node splitting first locates two entries that are furthest apart. That is, a pair of entries that would waste the largest area if they are put in the same group. These two rectangles are known as the seeds and the pair chosen tend to be small relative to others. Two groups are formed, each with one seed. For the remaining entries, each entry rectangle is used to calculate the area enlargement required in the covering rectangle of each group to include the entry. The difference of two area enlargements is calculated and the entry that has the maximum difference is selected as the next entry to be included into the group whose covering rectangle needs the least enlargement. As the selection is mainly based on the minimal enlargement of covering rectangles and the rectangle that has been enlarged before requires less expansion to include the next rectangle, it is quite often that a single covering rectangle is enlarged till the group has M–m+1 rectangles. The two resultant groups will respectively contain M–m+1 and m rectangles. The linear algorithm chooses the first two objects based on the separation between the objects in relation to the width of the entire group along the same dimension.

4.2.2. The R-tree Extensions

The coverage of covering rectangles and overlaps between them in the R-tree are affected by the objects are being partitioned into groups by its splitting algorithm. In [Gre89], Greene proposed a slightly different splitting algorithm. In her splitting algorithm, two most distant rectangles are selected and for each dimension, the separation is calculated. Each separation is normalized by dividing it with the interval of the covering rectangle on the same dimension, instead of by the total width of the entire group as in [Gut84]. Along the dimension with the largest normalized separation, rectangles are ordered on the lower coordinate. The list is then divided into two groups, with the first $\frac{(M+1)}{2}$ rectangles into the first group and the rest into the other.

The R*-Tree

Minimization of both coverage and overlaps is crucial to the performance of the R-tree. It is however impossible to minimize the two at the same time. A balancing criterion must be found such that the near optimal of both minimization can produce the best result. An additional optimization objective put forward in [BKS90] is the margin of the covering rectangles; squarish covering rectangles are preferred. Based on the fact that clustering rectangles with little variance of the lengths of the edges tend to reduce the area of the cluster's covering rectangle, the criterion that ensures the quadratic covering rectangles is used in the insertion and splitting algorithms of the improved R-tree, called the R^*-tree.

In the leaf nodes of the R^*-tree, a new record is inserted into the page whose entry covering rectangle if enlarged has the least overlap with other covering rectangles. A tie is resolved by choosing the entry whose rectangle needs the least area enlargement. However, in the internal nodes, an entry whose covering rectangle needs the least area enlargement is chosen to include the new record, and a tie is resolved by choosing the entry with the smallest resultant area. The improvement is particularly significant when both the query rectangles and data rectangles are small, and when the data is non-uniformly distributed. In the R^*-tree splitting algorithm, along each axis, the entries are sorted by the lower value, and also sorted by the upper value of the entry rectangles. For each sort, M–2m+2 distributions of splits are considered, where in kth ($1 \le k \le M-2m+2$) distribution, the first group contains the first (m–1)+k entries and the other group contains the remaining M–m–k entries. For each split, the total area, the sum of edges and the overlap-area of the two new covering rectangles are used to determine the split. Note that not all three can be minimized at the same time. In [BKS90], three selection criteria were proposed based on the minimum over one dimension, the minimum of the sum of the three values over one dimension or one sort, and the overall minimum. In the algorithm, the minimization of the edges is used.

Dynamic hierarchical spatial indexes are sensitive to the order of the insertion of data. A tree may behave differently for the same data set but with a different sequence of insertions. Data rectangles inserted previously may result in a bad split in R-tree after some insertions. Hence it may be worth to do some local reorganization, which is however expensive. The R-tree deletion algorithm provides reorganization of the tree to some extent, by forcing the entries in underflowed to be inserted from the root. The study in [BKS90] shows that the deletion and reinsertion can improve the R-tree quite significantly. Using the idea of reinsertion of the R-tree, Beckmann et al proposed a reinsertion algorithm when a node overflows. The reinsertion sorts the entries in decreasing order of the distance between the centroids of the rectangle and the covering rectangle and reinserts the first *p* (variable for tuning) entries. In some cases, the entries are reinserted back into the same node and hence a split is eventually necessary. The reinsertion will no doubt increase the storage utilization; but it can be fairly expensive when the tree is large. In the experiments conducted in [BKS90], the R^*-tree is found to be more efficient than some other variants, and the R-tree using linear splitting algorithm is substantially less efficient than the one with quadratic splitting algorithm. In general, the R^*-tree is an improvement over the R-tree at the expense of more expensive insertion.

The Buddy-Tree

The buddy-tree which can be considered as a compromise of the R-tree and the grid-file was proposed by Seeger et. al. in [SeK90]. It avoids the downward splitting of the K-D-B-tree, the overlapping problem of the R-tree and the dependency of structures upon the insertion of data. The buddy-tree generalizes the buddy system of the grid-file to organize correlated data efficiently, by bounding the data points tightly using the bounding rectangle concepts of the R-tree and organize the directory as in the R-tree. Like grid-files, the non-zero sized data have to be mapped into higher dimension.

The Packed R-Tree

To construct a tree with all the data available beforehand, the data can be pre-processed to produce a compact and query efficient tree. In [RoL84, RoL85], packed R-trees were introduced to minimize the coverage and overlap of rectangles by building an R-tree statically. It was shown that for point data, it is possible to partition points into groups such that the bounding rectangles of these groups do not overlap. However, to achieve zero overlap may require rotating the orientation of the entire database, which may not be possible or beneficial. Further, zero overlap is achievable only at the leaf level of the R-tree with static construction. For bounding rectangles associated with the non-leaf nodes, overlap is sometimes unavoidable. The main objective of the algorithm is to reduce the storage space, the coverage and overlap of rectangles, in order to improve the search efficiency.

The algorithm first orders the objects along one of the co-ordinate axes (e.g. X) using the smaller co-ordinate value for each bounding rectangle. The algorithm then chooses the first object from the list and using that object, selects the nearest M–1 objects to form a node, where M is the maximum number of objects or entries allowed in a page. The process is repeated till all objects are assigned to nodes. The bounding rectangle enclosing all objects in a leaf node becomes an object at the next higher level. These objects are once again ordered and assigned to nodes in the same way. The algorithm stops when there are less than M objects left; these remaining objects are assigned to the root node of the R-tree.

Although the algorithm [RoL84, RoL85] uses a *nearest* function to identify the next object to be included in the current node, this function is not defined. As far as the measure of distance is concerned, there are several ways to determine the distance between two objects, for example:

(1) the minimum distance between any two points on the boundary of each object;

(2) the distance between the centroids of the objects;

(3) either (1) or (2) with respect to the bounding rectangles of the objects;

To minimize the coverage of rectangles in a packed R-tree, the use of distance between the centroids of the bounding rectangles is a better measure than the two nearest points because:

(1) objects that are close together based on the latter measure of distance can still have large coverage, and

(2) the coverage of both *A* and *B* are determined based on the maximum and minimum values of the two objects, which are the boundaries of bounding rectangles.

A slightly different way of packing is to incorporate the enlargement criterion of the R-tree to select the next entry. That is, using the bounding rectangle covering all the entries selected so far, the next entry is the one that requires the least enlargement of the bounding rectangle to cover the new entry. In Figure 13, rectangle *b* requires the smallest bounding rectangle (BR) expansion, but rectangle *a* is the nearest to the node bounding rectangle. Hence, rectangle *b* will be selected with the above method, but rectangle *a* will be selected by the packed R-tree strategy.

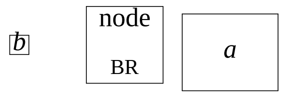


Figure 13 Selection of next rectangle

The R$^+$-Tree

The R$^+$-tree [SRF87] is a compromise between the R-tree and the K-D-B-tree [Rob81] and was proposed to overcome the problem of the overlapping covering rectangles of internal nodes of the R-tree.

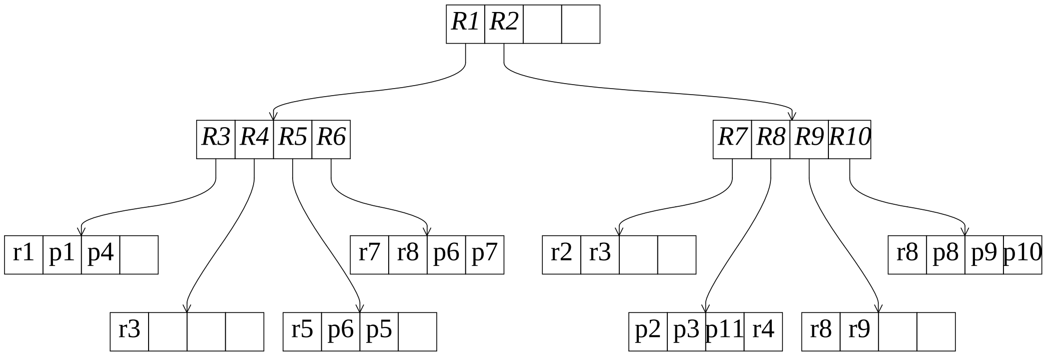
The R$^+$-tree structure is exactly the same as that of the R-tree, however the constraints are slightly different.

(1) Nodes of an R$^+$-tree are not guaranteed to be at least half filled.

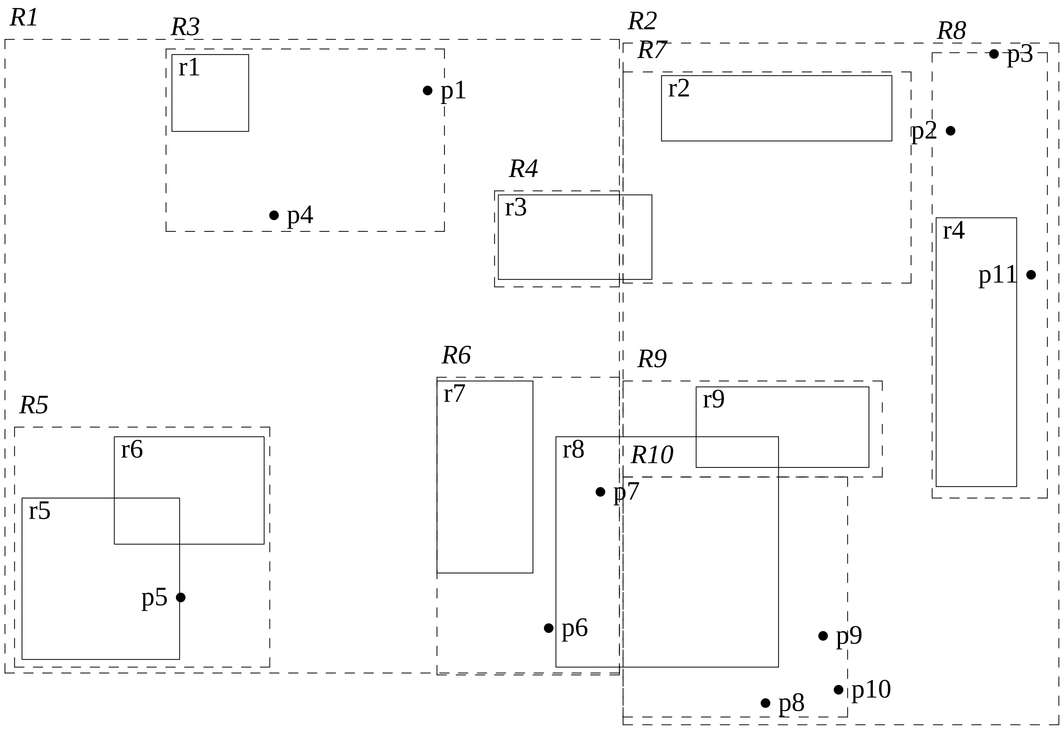
(2) The entries of any intermediate (internal) node do not overlap.

(3) An object identifier may be stored in more than one leaf node.

The duplication of object identifiers leads to the non-overlapping of entries. In a search, the subtrees are searched only if the corresponding covering rectangles intersect



(a) The directory of an R$^+$-tree



(b) A planar representation

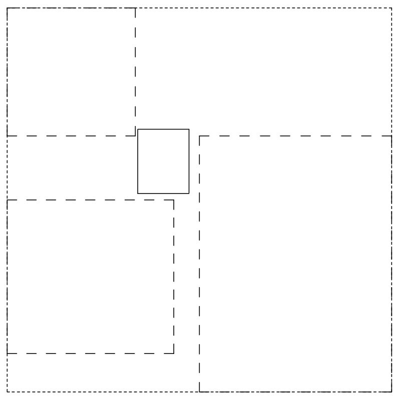
Figure 14 The structure of an R$^+$-tree the query region. The disjoint covering rectangles avoid the multiple search paths of the R-tree for point queries. For the space in Figure 14, only one path is traversed to search for all objects that contain point *p7*; whereas for the R-tree, two search paths exist. However, for certain query rectangles, searching the R^+-tree is more expensive than searching the R-tree. For example, suppose the query region is the left half of object *r8*. To retrieve all objects that intersect the query region using the R-tree, two leaf nodes have to be searched, respectively through *R5* and *R8*, and it incurs five page accesses. To evaluate such a query, three leaf nodes of the R^+-tree have to be searched, respectively through *R6*, *R9*, and *R10*, and a total of six page accesses is incurred.

To insert an object, multiple paths may be traversed. At a node, the subtrees of all entries with covering rectangles that intersect with the object bounding rectangle must be traversed. On reaching the leaf nodes, the object identifier will be stored in the leaf nodes; multiple leaf nodes may store the same object identifier.

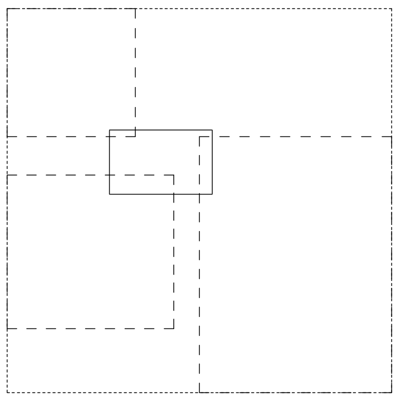
Three cases of insertions need to be handled with care [Gun88, Ooi88]. The first is when an object is inserted into a node where the covering rectangles of all entries do not intersect with the object bounding rectangle (see Figure 15a). The second is when the bounding rectangle of the new object only partially intersects with the bounding rectangles of entries (as shown in Figure 15b); this requires the bounding rectangle to be updated to include the new object bounding rectangle. Both cases must be handled properly such that the coverage of bounding rectangles and duplication of objects could be minimized.

The third case is more serious in that the covering rectangles of some entries can prevent each other from expanding to include the new object. In other words, some space ("dead space") within the current node cannot be covered by any of the covering rectangles of the entries in the node. If the new object occupies such a region, it cannot be fully covered by the entries. For example, the solid rectangle in Figure 16 cannot be covered by the bounding rectangles (dashed rectangles). To avoid this situation, it is necessary to look ahead to ensure that no dead space will result when finding the entries to include an object. Alternatively, the criterion proposed by Guttman [Gut84] can be used to select the covering rectangles to include a new node. When a new object cannot be fully covered, one or more of the covering rectangles are split. This means that the split may cause the children of the entries to be split as well, which may further degrade the storage efficiency.

During an insertion, if a leaf node is full and a split is necessary, the split attempts to reduce the identifier duplications. Similar to the K-D-B-tree, the split of a leaf node may propagate upwards to the root of the tree and the split of a non-leaf node may propagate downwards to the leaves. The split of a node involves finding a partitioning



(a)



(b)

Notation: current bounding rectangle parent covering rectangle

query rectangle

Figure 15 Objects covering in R^+-trees

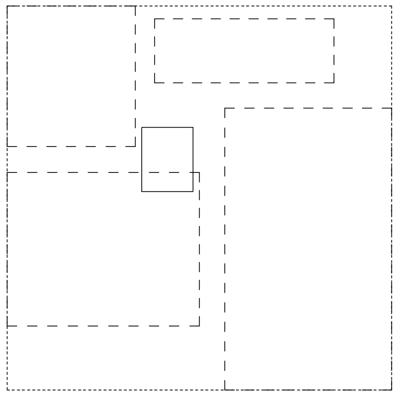


Figure 16 The deadlock: The input rectangle cannot be covered

hyperplane to divide the original space into two. The selection of a partitioning hyperplane was suggested to be based on the following four criteria: the clustering of entry rectangles, minimal total x- and y-displacement, minimal total space coverage of two new subspaces, and minimal number of rectangle splits. While the first three criteria aim to reduce search by tightening the coverage, the fourth criterion confines the height expansion of the tree. The fourth criterion can only minimize the number of covering rectangles of the next lower level that must be split as a consequence. It cannot guarantee that the total number of rectangles being split is minimal. Note that all four criteria cannot possibly be satisfied at the same time.

The problem of overlapping rectangles of the R-tree is overcome, the R^+-tree however inherits some problems of the K-D-B-tree [Rob81] and the Matsuyama's kd-tree [MHN84]. Like K-D-B-tree, partitioning a covering rectangle may cause the covering rectangles in the descendant subtree to be partitioned as well. Frequent downward splits tend to partition the already under populated nodes, and hence the nodes in an R^+-tree may contain less than $M/2$ entries. Object identifiers are duplicated in the leaf nodes, the extent of duplication is dependent on the spatial distribution and the size of the objects. To delete an object, it is necessary to delete all identifiers that refer to that object. Deletion may necessitate major reorganization of the tree.

In [Gre89], a performance study of R-trees and R^+-trees was conducted. In the comparison between R-trees and R^+-trees, it is found that the R^+-tree requires much more splits, especially for large data objects, but lesser splits for smaller data objects. For a uniform data distribution of square rectangles that fully covers the map space, 30% of the objects are duplicated. Interestingly, the results show that for the case where the coverage is 100% and the objects are long and narrow along X dimension, the duplication decreases. This is likely due to the better grouping achieved along *X* axis. In general, the query efficiency tests show that R^+-trees perform better for smaller objects and slightly worse off for larger objects. The study in fact exhibits similar pattern of results to that of the kd-trees extended using the overlapping approach and the non-overlapping approach [Ooi90].

The Cell Tree

Based on the binary space partitioning trees [FKN80], Gunther [Gun88] proposed the *cell tree* to alleviate the overlapping bounding rectangle problems of R-trees and the "dead space" problems of R^+-trees. Binary space partitioning trees are binary trees that represent a recursive subdivision of a space into subspaces. The cell-tree is a height-balanced tree. The partitioning hyperplanes in a cell-tree may not be parallel to any axis, and as a result, the unpartitioned subspaces are polyhedral (see Figure 17). In a cell-tree internal node, the bounding polygon of an entry is a convex polyhedron. The sons of each node form a binary space partition of the node. Partitions do not overlap. The fact that polyhedra may require different number of points to represent is difficult for the cell-tree to set a lower bound on the number of entries. An insertion may cause a node to overflow. When a page split is not possible, a node may occupy more than one page.

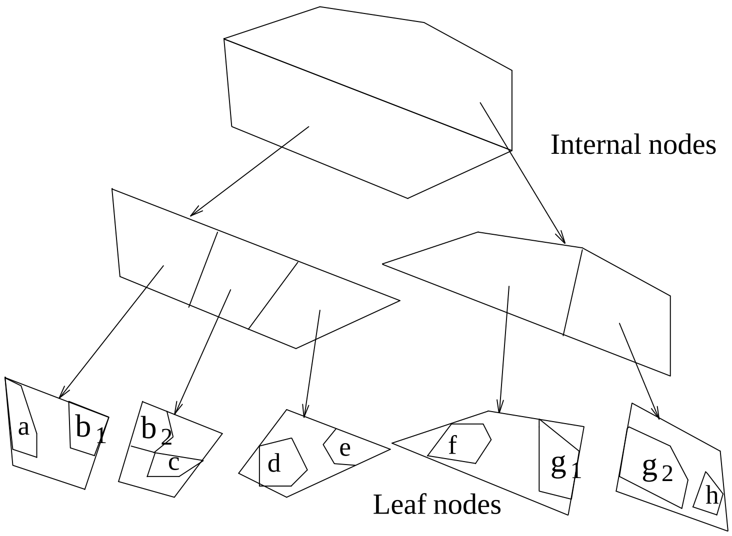


Figure 17 The structure of a cell tree

Spatial objects are no longer bounded by bounding rectangle, but by a convex polyhedron. A convex polyhedral cover of an object is composed of a set of polyhedra to better approximate irregularly shaped spatial objects. Like the R^+-tree, a spatial object being represented may be stored in more than one leaf node.

One major problem with the indexes employing object duplication techniques is that each new object may be divided into multiple pieces in order to store them in a tree where internal node bounding polygons do not overlap. The fragmentation effect is even more serious when the database becomes more populated. Each split of a node leads to a decrease in the node data space but to an increase in the number of nodes per object. To overcome the fragmentation and duplication problems, Gunther and Noltemeier [GuN91] proposed to store oversized objects which may greatly increase the number of object identifiers being stored in the leaf nodes in separate "oversize shelves". These oversize shelves are data nodes linked to internal nodes in the cell-tree, in one way, causing the tree to be not height-balanced. The placement of a new object in the subtree or oversize shelf requires some optimization. The oversize page shelf can be overflowed and a split on this shelf is necessary. We did some experimentation before on the R-tree in such a way that the oversized objects are stored in an internal node as long as possible, and these oversized objects are pushed down to next level only when the node has M subtree entries. We found that this gives a better storage efficiency, but the improvement in query efficiency is not significant. This is due to the fact that by storing data objects in internal nodes, the height of the tree is increased.

4.3. Quad-tree Based Structures

The quad-CIF-tree [FKS81, Sam84] (where CIF denotes Caltech Intermediate Form) was proposed for representing a set of small rectangles for VLSI applications. It is organized in a way similar to the region quad-tree. A region is recursively partitioned until the resulting quadrants do not contain any rectangle. During the subdivision, all rectangles that intersect with either of the two partitioning lines are associated with the partitioning lines. The rectangles that are associated with a quadrant must not belong to any ancestor quadrant. It is assumed that no two rectangles overlap.

An intersection search performed on the quad-CIF-tree would begin with the root node, and examine the rectangles associated with it. The search examines only the child nodes of quadrants that intersect the query region.

To insert a rectangle, each subdivision node (quadrant) is checked. If one of the axes intersects with the rectangle, the rectangle is inserted at that node. Otherwise, the child node whose quadrant contains the rectangle is searched. If the quadrant does not have any child and the rectangle has not been inserted, then the process of recursive subdivision of quadrants is required. Deletion of a node is a counterpart of the node splitting during insertion, where tree collapsing is involved.

In contrast to the quad-CIF-tree representation, a point representing a rectangle may be used to store rectangles in a PR quad-tree [Sha86]. A problem with this representation is that practically the entire tree has to be searched for intersection queries, which is due to the fact that rectangles may span over any parts of the space.

4.4. Cell Methods based on Dynamic Hashing

Both extendible hashing [FNP79] and linear hashing [KrS86, Lar78] lend themselves to an adaptable cell method for organizing *k*-dimensional objects. The grid file [HiN83, NHS81, NHS84] and the EXtendible CELL (EXCELL) method [Tam82a, Tam82b] are extensions of dynamic hashed organizations [FNP79] incorporating a multi-dimensional file organization for multi-attribute point data.

The Grid File

The *grid file* structure proposed in [HiN83, NHS84] consists of two basic structures: *k* linear *scales* and a *k*-dimensional *directory* (see Figure 18). The fundamental idea is to partition a *k*-dimensional space according to an orthogonal grid. The grid on a *k*-dimensional data space is defined as scales which are represented by *k* one-dimensional arrays. Each boundary in a scale forms a (*k*-1)-dimensional hyperplane that cuts the data space into two subspaces. Boundaries form *k*-dimensional unpartitioned rectangular subspaces, which are represented by a *k*-dimensional array known as the *grid directory*. The correspondence between directory entries and grid cells (blocks) is one-to-one. Each

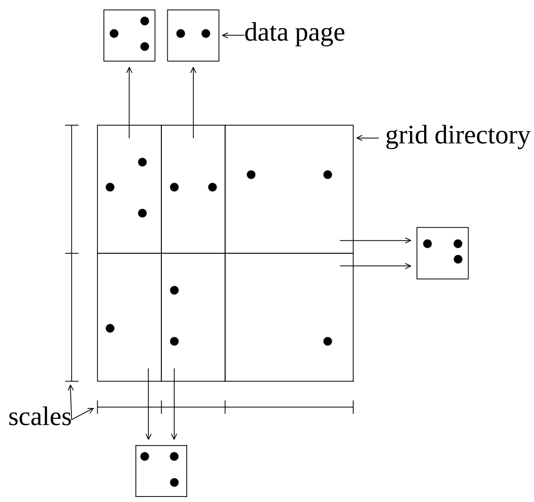


Figure 18 The grid file layout

grid cell in the grid directory contains the address of a secondary page, the *data page*, where the data objects that are within the grid cell are stored. As the structure does not have the constraint that each grid cell must at least contain *m* objects, a data page is allowed to store objects from several grid cells as long as the union of these grid cells together form a rectangular rectangle, which is known as the *storage region*. These regions are pairwise disjoint, and together they span the data space. For most applications, the size of the directory dictates that it be stored on secondary storage, however, the scales are much smaller and may be cached in main memory.

Figure 19 illustrates a three dimensional grid object space. A three-dimensional array, *dir*(1..3, 1..3, 1..2), is required to store the grid entries, and the description of an entry may be obtained once the scales are known. For example, the description of the grid cell with $x_3 \times y_1 \times z_2$ subspace is stored in *dir*(3, 1, 2).

Like other tree structures, splitting and merging of data pages are respectively required during insertion and deletion. Insertion of an object entails determining the correct grid cell and fetching the corresponding page followed by a simple insertion if the data page is not full. In the case where the page is full, a split is required. The split is simple if the storage region covers more than one grid cell and not all the data in the region fall within the same cell; the grid cells are allocated to the existing data page and a new page with the data objects distributed accordingly. However, if the page region covers only one grid cell or the data of a region fall within only one cell, then the grid has to be extended by a (*k-1*)-dimensional hyperplane that partitions the storage region into two subspaces. A new boundary is inserted into one of the k grid-scales to maintain the one-to-one correspondence between the grid and the grid directory, a (*k*-1)-dimensional cross-section is added into the grid directory. The resulting two storage regions are disjoint and, to each region a corresponding data page is attached. The objects stored in the overflowing page are distributed among the two pages, one new and one existing page. Other grid cells that are partitioned by the new hyperplane are unaffected since

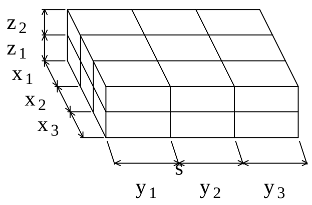


Figure 19 A three-dimensional grid object space

both parts of the old grid cell will now be sharing the same data page.

Deletions may cause occupancy of a storage region to fall below an acceptable level, and these trigger merging operations. When the joint occupancy of a storage region whose records have been deleted and its adjacent storage region drops below a certain threshold, the data pages are merged into one. In [NHS84], based on the average bucket occupancy obtained from the simulation studies, 70% is suggested to be an appropriate of the resulting bucket. Two different methods were proposed for merging, the *neighbor* system and the *buddy* system. The neighbor system allows two data pages whose storage regions are adjacent to merge so long as the new storage region remains rectangular; this may lead to "dead space" where neighboring pages prevent any merging for a particular underpopulated page. For example, the data page with region *A* in Figure 19, if underflows, cannot be merged with any data pages of neighboring regions. A more restrictive merging policy like the buddy system is required to prevent the dead space. For the buddy system, two pages can be merged provided their storage regions can be obtained from the subsequent larger storage region using the splitting process. However, total elimination of dead space for a *k*-dimensional space is not always possible.

The merging process will make the boundary along the two old pages redundant, when there are no storage regions adjacent to the boundary. Consider Figure 21, the boundary *s* becomes redundant if all the storage regions that span over y_1 and y_2 intervals along dimension Y intersect with *s*. In this case, the redundant boundary is removed from its scale and the one-to-one correspondence is maintained by removing the redundant entries from the grid directory.

In [NiH85], the grid file was proposed as a means for spatial indexing of non-point objects. To index *k*-dimensional data objects, mapping from a *k*-dimensional space to a n*k*-dimensional space where objects exist as points is necessary. One disadvantage of the mapping scheme is that the higher dimensional space means it is harder to perform

Figure 19 Dead space introduced by the neighbor merging system directory splitting [WhK85]. To index a rectangle, it is represented as (*cx, cy, dx, dy*), where (*cx,cy*) is the centroid of the object and (*dx, dy*) are the extensions of the object from the centroid [HiN83]. The (*cx, cy, dx, dy*) representation causes objects to cluster close to X-axis, while objects cluster on top of $x = y$ for (x_1, x_2, y_1, y_2) represention. For ease of grid partitioning, the former representation is therefore preferred [Nie95].

For an object (*cx, cy, dx, dy*) to intersect with the query region (*qcx, qcy, qdx, qdy*), the following conditions must be satisfied:

$$
\begin{aligned}
&cx - dx \le qcx + qdx \text{ and}\\
&cx + dx \ge qcx - qdx \text{ and}\\
&cy - dy \le qcy + qdy \text{ and}\\
&cy + dy \ge qcy - qdy.
\end{aligned}
$$

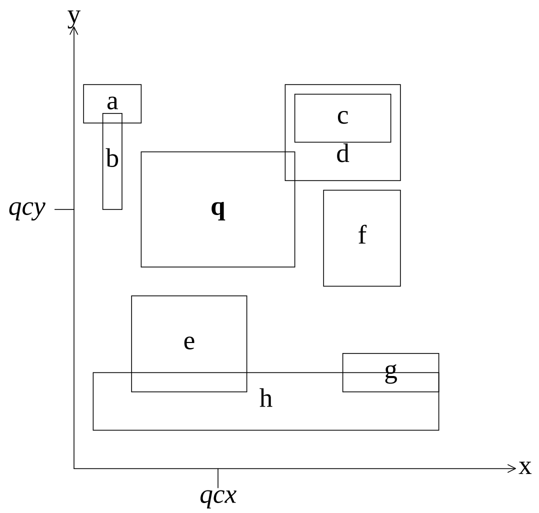
Consider Figure 20a, where rectangle q is the query rectangle. The intersection search region on cx-dx hyperplane, the shaded region in Figure 20b, is obtained by the first two inequality equations of the above intersection condition. Note that the search region can be very large if the global space is large and the largest rectangle extension along the X axis is not defined. In Figure 20b, the known upper bound, *udx*, for any rectangle extension along the X axis, reduces the search region to the enclosed shaded region. The same argument applies for the other co-ordinate. Objects that fall in both search regions satisfy the intersection condition.

It is easier to understand how the search is performed from a two-dimensional view point rather than from a four-dimensional view point. Suppose the global two-dimensional space of a database is defined and is represented as (0, 0) and (*gx*, *gy*). Further, suppose the maximum extensions of objects are known, and let them be *udx* and *udy* respectively. Then the search region for the query defined by two corners, (*qx1, qy1*) and (*qx2, qy2*) is constrained by the following two equations:

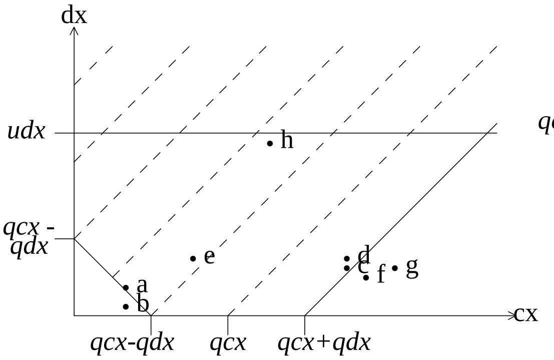
$$
\begin{aligned}
&\text{MAX}(\frac{qx1}{2}, \text{qx1 - udx}) < \text{x} < \text{MIN}(\frac{gx + qx2}{2}, \text{qx2 + udx})\\
&\text{MAX}(\frac{qy1}{2}, \text{qy1 - udy}) < \text{y} < \text{MIN}(\frac{gy + qy2}{2}, \text{qy2 + udy})
\end{aligned}
$$

where MAX and MIN functions respectively return the maximum and the minimum of their arguments. Figure 21 illustrates the search space defined by the above condition. Objects whose centroids are in the shaded region will not intersect with the query region. However, not objects whose centroids are in the unshaded region will intersect with the query region.

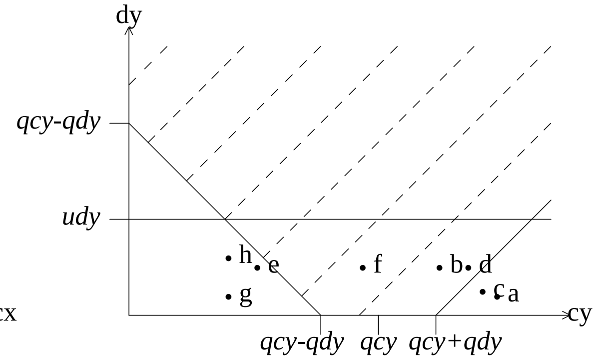
The mapping of regions from a *k*-dimensional space to points in a n*k*-dimensional space undesirably changes the spatial neighborhood properties. Regions that are spatially close in a *k*-dimensional space may be far apart when they are represented as points in an



(a) The distribution of data objects



(b) Search regions on cx-dx hyperplane



(c) Search region on cy-dy hyperplane

Figure 20 Intersection search region in the grid file

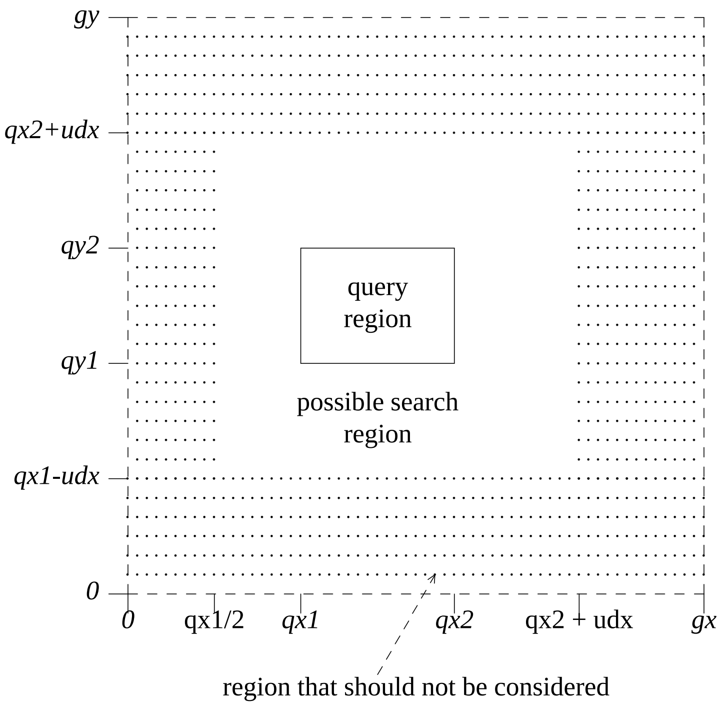


Figure 21 A two-dimensional view of a search space

nk-dimensional space. Consequently, the intersection search may not be efficient.

The EXCELL Method

The EXCELL method of Tamminen [Tam82a, Tam82b] is an independent work that uses the same approach as that of the grid file. The major difference between the two is the way in which the search space is partitioned. Instead of allowing a space to be partitioned freely, the EXCELL method requires that all partitioning lines be equi-distant. When an interval is partitioned, other intervals on the same dimension must be partitioned as well. All grid cells cover the same amount of space, and the size of the grid directory is doubled with every split. The EXCELL method simplifies the grid partitioning operations at the expense of requiring larger directories.

In [Tam83], Tamminen suggested a hierarchical EXCELL method to alleviate the problem of large grid directories. The approach is similar to the multi-level grid file implementation proposed in [WhK85], in which each cell may correspond to a data page or a sub-directory. However in Tamminen's hierarchical EXCELL method, the maximum depth of the hierarchy is fixed and when a data page overflows, it will not be further split and overflow pages are used instead. In [TaS82], the EXCELL method was suggested for indexing non-zero sized objects by duplicating object identifiers.

The Multi-Level Grid Files

The grid file structure [NHS84] was originally designed to guarantee two disk accesses for exact match queries, one to access the directory and the other to access the data page.

The "two disk access" property can only be ensured if the directory is stored as an array and all grid cells are of the same size. However, with such an implementation, the size of the directory is doubled whenever a new boundary is introduced. Most of these directory entries correspond to empty grid cells, i.e. they do not contain any data objects. Simulated results [NHS84] indicate that the size of the directory grows approximately linearly with the size of the file. To alleviate this problem, a multi-level directory [BIM90, Hin85, HSW90, Fre87, WhK85] where grid cells are organized in a hierarchical structure was suggested. A multi-level grid file called the *generalized grid file* (GGF) was proposed in [BIM90]. The GGF behaves like a B^+-tree for a single dimensional data. With such a hierarchical structure, the property of "two disk accesses" [NHS84] for exact-match queries is no longer valid.

In [HSW88a, HSW88b], the *twin grid file access method* was proposed to improve the low storage utilization of grid files by using two grid files, called the primary grid file and the secondary grid file. In [HSW88b], it was assumed that the storage space consumed by the grid directories is fairly small as compared to the space required for data buckets, hence only the space for data buckets is optimized. The basic idea of the twin grid file is to distribute data points among the two files which span over the same entire data space. In an insertion, the point is firstly inserted into the primary grid file. If the bucket overflows, a point has to to be transferred from the primary grid file to the secondary grid file if it has space for the point. Otherwise, a split is required. Deletions may cause a bucket in one of the grid file to underflow, shifting of points are required if two corresponding buckets cannot be merged. Note that the restructuring may already have occurred within a single file, as in conventional grid files. For example, a data point which if inserted into the primary file would cause a bad split is inserted into the

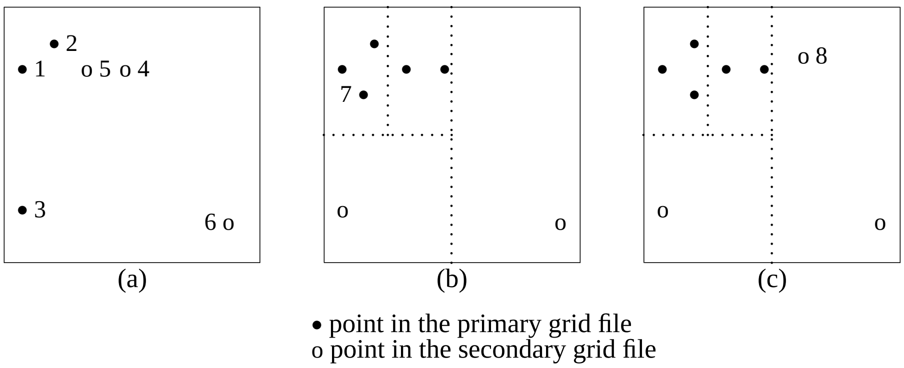


Figure 22 Twin grid file organization

secondary grid file. However, with further insertions, the points that are previously distributed into the secondary grid file may give a better storage utilization if they are brought back to the primary grid file for a split. Hence, points may be transferred from the secondary grid file to the primary grid file, or vice versa, after each insertion or deletion. In [HSW88b], the primary grid file is favored over the secondary grid file if the storage of a point does not increase the number of data buckets. Their experimental results show that only about 15 of the points are stored in the secondary file. Consider Figure 22; let the dotted boundaries be those from the primary grid file and the page capacity be 3. Figure 22a illustrates an ideal distribution where the number of buckets required is minimal. However, as object 7 is inserted, a redistribution is required (see Figure 22b). Object 3 is moved from the primary grid file to the secondary grid file and objects 4 and 5 are brought forward to the primary grid file. Figure 22c is a case in point where simple insertion can be achieved (object 8). The main objective of object migrations is to minimize the number of buckets used to store a given set of points. The high space utilization is achieved at the expense of more expensive insertions and deletions which incur partial reorganization. Indeed, the experimental results reported in [HSW88b] show that the number of page accesses during insertions in dynamic twin grid files can be twice as many as that of the conventional grid file. To answer a query, whether it is a point query or a region query, two grid files have to be searched. The savings in data buckets may hence cost more in search time.

The Multi-layer Grid File

To improve the search performance of the grid file, a *multi-layer grid file* which avoids object mapping was proposed in [SiW88]. In such a structure, a map space may consist of several grid files that cover the same space. When a grid file is partitioned, all objects that are not cut by the partitioning hyperplane are distributed among the two new subspaces and objects that are cut are stored in the next layer grid file. There may be several layers of grid files that store unpartitioned objects, and each layer has different partitioning hyperplanes. At the maximal layer, the objects are clipped if they intersect the partitioning hyperplane. Figure 23 illustrates the 3-layer grid files: objects with solid lines and points are stored in the first layer, objects with dashed lines in the second and objects with dotted lines in the third. Using multiple layers, the number of objects being clipped is reduced as compared to the clipping technique in a single layer grid file.

Although the multi-layer grid file avoids the mapping of spatial objects into points in a higher dimensional space, it suffers from expensive directory overhead and undesirably low storage utilization (between 50% and 60%) [HSW90]. Further, searching for spatial objects intersecting a query region over multiple grid files is not efficient.

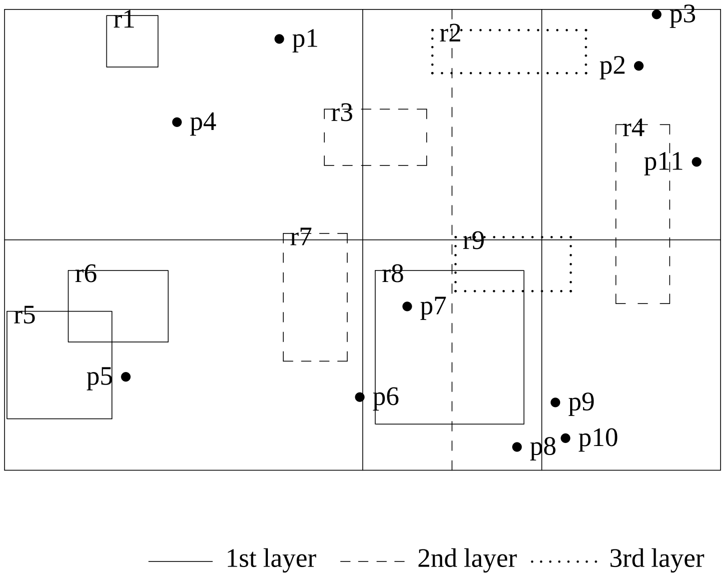


Figure 23 Multi-layer grid files

The R-file

As an attempt to improve the performance of grid files, Hutflesz et al [HSW90] proposed an alternative scheme called *the R-file* based on the concept of multi-layer grid files. The R-file is different from the multi-layer grid file in that the R-file has only one layer and is intended for non-zero sized objects. In the R-file, cells are partitioned using the partitioning strategy of the grid file [NHS84] and a cell is split when overflowed. In order for cells to tightly contain the spatial objects, cells are partitioned recursively by repeated halving till the smallest cell that encloses the spatial objects is obtained. Spatial objects that are totally contained in a cell are stored in its corresponding data page, and those that intersect the partitioning line are stored in the original cell. If the number of spatial objects that intersect a partitioning is more than what can be stored in a data page, partitioning line along the other dimensions will be used. If all records lie on the cross point of partitioning lines, they cannot be partitioned by any partitioning lines, and in such a case, a chain of buckets is used.

After a split, the original cell and the two new cells overlap and to keep the directory small, empty cells are not maintained. After a split, both the original and new cells have almost the same number of spatial objects. Figure 24 illustrates a case in point. Even so, a high number of cells will be inspected for intersection queries, especially those original large cells. The fact that spatial objects stored in the original unpartitioned cells tend to intersect the partitioning lines of the cells suggests the clustering property of these objects. In order to make intersection search more efficient, two extra values that bound the objects in the partitioning dimension are kept with the original cells. Due to the overlapping cells, the directory is potentially large. To avoid storing the cell boundaries, a z-ordering scheme [Ore86] is used to number the cells. With such a scheme, cells are partitioned cyclically. For a split not according to the cycle, additional information is stored so that one dimension due to non-split can be skipped. For each cell, the directory stores the cell number, the bounding interval, and the data bucket reference. The experiments conducted in [HSW90] strongly indicate that the bounding information leads to substantial saving of page accesses.

PLOP-Hashing

In [KrS88], a grid file extension was proposed for the storage of non-zero sized objects. The method is a multi-dimensional dynamic hashing scheme based on Piecewise Linear Order Preserving (PLOP) hashing. Like the grid file [NHS84], the data space is partitioned by an orthogonal grid. However, instead of using *k* arrays to store scales that define partitioning hyperplanes, *k* binary trees are to represent the linear scales. Each internal node of a binary tree stores a (*k*-1)-dimensional partitioning hyperplane. Each leaf node of a binary tree is associated with a *k*-dimensional subspace (a slice), where the interval along its associated axis is a sub-interval and the other *k*-1 intervals assume the intervals of the global space. Each slice is addressed by an index *i* stored in its leaf node.

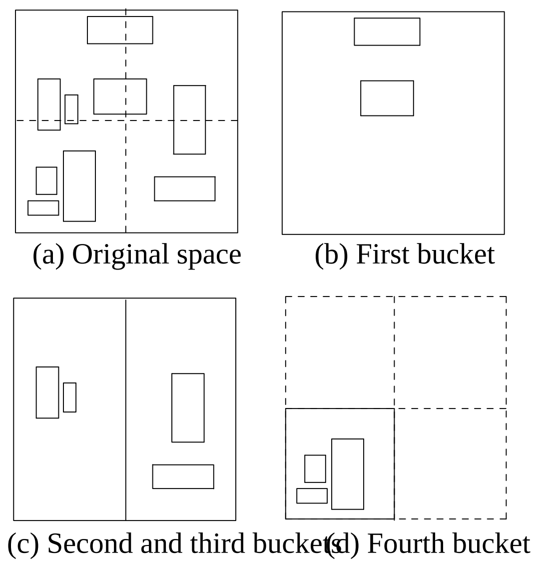


Figure 24 The R-file

To each cell, a page is allocated to store all points that fall in the unpartitioned subspace. From the indexes stored in *k* binary trees, the address of a page can be computed. Adopting the bounding scheme similar to that of [OMS87], two extra values are stored in a leaf node to bound the objects whose centroids are in the corresponding slice along the axis that the binary tree is associated with. Hence, an object is inserted into the grid cell that contains it centroid. The regions defined by the two extra values may overlap and they will be used for intersection search. The implementation of PLOP-hashing presented in [SeK88] stores the binary trees in main memory.

The file organizations based on hashing are generally designed for multi-dimensional point data. To use them for spatial indexing, the mapping of objects from *k*-dimensional space to n*k*-dimensional space or duplication of objects identifiers are generally required. Indexing in a parameter space is not efficient for general spatial query retrievals [Gut84, WhK85].

4.5. Spatial Objects Ordering

Existing DBMS supports efficient one-dimensional indexes and provides fast access to one-dimensional data. If multi-dimensional objects can be converted to one-dimensional objects, such indexes can be used directly without alteration. The mapping functions used in mapping must preserve the proximity between data well enough in order to yield reasonably good spatial search. The idea is to assign a number to each of representative grid in a space and these numbers are then used to obtain a representative number for the spatial objects. In this section, we shall review three different mappings.

Ordering Based on Quad-Tree Quadrants

A mapping which is in principle similar to the quad-CIF-tree approach was proposed by Abel and Smith [AbS83]. A quad-tree divides a space into four equal sized square subspaces. For each subspace of a quad-tree, a unique numeric key of base 5 is attached. All objects falling within a given subspace are assigned the subspace's key. The key *k* for a subspace of level *h* (> 1) can be derived from the key (*k'*) of the ancestor subspace by the following formula:

$$k \equiv \begin{cases} k' + 5^{m-h} & \text{if} \quad k \textit{ is the SW son of } k' \\ k' + 2*5^{m-h} & \text{if} \quad k \textit{ is the NW son of } k' \\ k' + 3*5^{m-h} & \text{if} \quad k \textit{ is the SE son of } k' \\ k' + 4*5^{m-h} & \text{if} \quad k \textit{ is the NE son of } k' \end{cases}$$

Here *m* is an arbitrary maximum number of levels in decomposition, which is greater than *h*. The global space has 5^m as the key.

Figure 25 illustrates an example of key assignment (base 5), where the maximum level of decomposition is 3. One can notice that, when the locational keys of the same level are traced, the ordering is a form of N- or Z-ordering.

To assign a key to a rectangle, the smallest block which completely covers the rectangle is used. An inherent problem of such an assignment is that an object bounding

Figure 25 Assignment of locational keys

Figure 26 Assignment of covering nodes rectangle may be very much smaller (as a consequence of the bounding rectangle spanning one or more subspace divisions) than the associated quadrant. To alleviate this problem, a decomposition technique [AbS84] is used, where a rectangle may be represented by up to four adjacent quadrants. Rectangles B and C in Figure 26 illustrate the cases where two and four quadrants are used: keys 110, 130, 123 and 141 for rectangle B, and keys 142 and 144 for rectangle C. By associating each rectangle with a collection of quadrants, a better approximation of a rectangle is achieved. This form of representation requires an object identifier to be stored in multiple locations. However, even if this approach is adopted, the size of the representative quadrant may still be much larger than the size of the object's bounding rectangle. Consider the rectangles A and B in Figure 26. A B^+-tree is used to index the objects based on their associated locational keys. For an intersection search, all quadrants that intersect the query region have to be scanned. The major advantage of the use of the locational key is that B^+-tree structures are widely supported by conventional DBMSs.

Ordering Based on Space-Filling Curves

Methods that use Peano space-filling curves [Pea90] to map *k*-dimensional space onto one-dimensional objects in its native space had also been proposed in [OrM84]. The idea is to transform each *k*-dimensional object to a set of line segments. A grid cell is assigned a unique value, and they are ordered using some space-filling curve strategies. The ordering of grid cells in [Ore86] is termed Z-ordering. A space consists of $2^m \times 2^m$ grid cells, and a rectangle is described by $[x_1, x_2]$ and $[y_1, y_2]$. A grid cell is referred to as an *element* and each of these elements has a unique z-value. The z-value of an element is a binary value obtained by interleaving the binary bits of the descriptive bits of each dimension. For example, an element with [011, 100] ([3, 4]) is addressed by 011010 (26), where the first bit of the z-value is the first bit of the X co-ordinate, the second bit of the z-value is the first bit of the Y co-ordinate, and so forth. The z values of the elements trace out the path shown in Figure 27.

For a region containing more than one grid cell, the common first n_i bits, along each dimension i are determined. In a two-dimensional space, all points lying inside the region have coordinates with the same n_x (along X axis) and n_y (along y axis) bit prefixes. The z-value of the region is constructed by interleaving the n_x and n_y bits. For example, a two-dimensional region defined by 010 and 011 along X axis, and 110 and 111 along Y axis has 01 and 11 as common prefixes for the X and the Y axes respectively. The z-value for the region is therefore 0111 and this bit string uniquely identifies the region. However, if a common prefix cannot be obtained, then a set of z-values is used. A rectangle is represented as a set of z-values in its "native" space, each corresponding to a cell that the rectangle completely covers. These values are not

Figure 27 Spatial interpretation of z order (Peano curve)

contiguous in the one-dimensional space. This occurs when a rectangle spans across grid cells of higher z-order (considering leftmost binary bits alone). The assignment of locational keys to spatial objects is a mapping of objects from a *k*-dimensional space to one-dimensional objects in its original space. The implementation is straightforward and conventional indexes can be used to index the objects.

In query processing, a query rectangle is transformed into a set of query intervals in a one-dimensional space. A data object is contained in the query rectangle if and only if the data object falls in all of the query intervals. Overlapping rectangles can be detected using the so-called spatial join [Ore86], which is a simple extension of the natural join. In such a join, each spatial object is transformed into a set of one-dimensional intervals, and for each object of an external join operand (object in the outer for loop), the objects of the internal join operand (the inner loop) are checked if one z-value is a prefix for the other. Each resulting candidate is a pair of spatial objects that are likely to overlap, and then a real spatial testing is performed. In [Ore90], the performance of *native* space implementation of the overlap query was shown to be better than that of the *parameter* space implementation in the context of z-order based indexing methods. The native space implementation transforms *k*-dimensional data objects into one-dimensional objects embedded in their native space, while the parameter space implementation maps *k*-dimensional *n* vertices objects to *nk*-dimensional point objects which may then be linearized.

Based on the concept of parameter space indexing in [Ore90], a method called DOT (DOuble Transformation) [FaR91] was proposed. The approach has two transformations:

first, a covering rectangle of an object in a *k*-dimensional space is mapped into a point in a *nk*-dimensional space; second, the n*k*-dimensional point is then mapped onto a point in a one-dimensional space using the distance preserving mapping concept. The second transformation can be any mapping, such as Hilbert [Hil91] or Peano [Pea90] mapping. The Hilbert curve involves rotation and reflection in its basic pattern (see Figure 28), which accounts for harder computation when compared to Peano curve. However, Hilbert curve was shown to produce a more efficient result [FaR91]. A separate study in clustering using different space filling curves was conducted in [Jag90b]; the experimental results suggest that Hilbert mapping is more superior to gray code and Peano mappings.

A point query is firstly transformed into a range query with a query rectangle in a two-dimensional space. The query rectangle is then transformed into a set of range queries in the final one-dimensional space. Data points fall within the intervals are the possible candidates.

The main motivation for spatial ordering methods is that the software and theory developed for conventional attribute-based searching can be readily used to support these methods. The transformation of data from one dimension to another may cause the loss of spatial proximity between data and poor intersection search as a result. The selection of effective distance preserving function is important.

Figure 28 Hilbert curve

Ordering Based on Grid Partitioning

The constraint that storage regions must be rectangular causes many storage regions to be much bigger than their actual data space. Interpolation-based grid files [Bur83] avoid this problem by using an explicit representation of data space so that there is always only one directory entry per data bucket.

The Balanced And Nested Grid (BANG) file [Fre87] is an interpolation-based grid file which is however different from the original grid file in that it allows two subspaces to intersect. The BANG file divides the data space into a hierarchy of sets of notational grid regions. Each of these grid regions can be identified by a unique pair (r, l), where r is the region number, and l is the granularity or level number. A space is obtained by recursively halving along some selected dimensions. That is, the level of the hierarchy of grid regions is generated from the previous higher level by partitioning along a selected dimension. For example, the region (0,0) represents the whole data space and after its partitioning, two subspaces (0,1) and (1,1) are obtained (see Figure 29a). In general, a grid region r at level l is partitioned into two uniquely numbered grid regions at level $l+1$. The left hand side region (or the lower region for partitioning performed on dimension Y) uses the old number r, and the right hand side region has the number $r+2^{l-1}$. The pair (r, l) forms a unique number for each grid region. New numbers can be easily obtained by extending the binary representation by one additional most significant bit. This representation enables non-regular partitioning and supports varying levels of granularity; an example is illustrated in Figure 29b. The order of these numbers provides a trace of z-ordering, and due to such fact that we review the technique in the current section.

In order to accurately reflect the clustering of points in a data space, the data space in the BANG file is allowed to contain concavities and hence it is not always rectangular. Suppose we initially have a data space S defining a block region R_1. After some insertions, the data page of R_1 overflows, which requires the partitioning of the grid cell. Suppose such a grid region is partitioned into 4 sub-grid regions and one of them, say R_2, contains almost half of the data items. The data items have been partitioned into two groups: one in R_2 and the other in R_1. This results in R_2 being *nested* or *enclosed* within R_1. For efficient query processing, the actual space of R_1 has to be derived. In this case, it is $S - S_2$, where S_2 is defined by R_2. Figure 30 illustrates such an example. In general, each data space in the BANG file is defined by a set of block regions where one region encloses all other regions. The enclosed regions represent substractions from the enclosing region. The testing during searching can be very expensive when the the number of rectangles representing substractons is large. Further, covering regions formed by grids may also be much bigger than the minimum covering regions. These two issues will affect the performance of query efficiency.

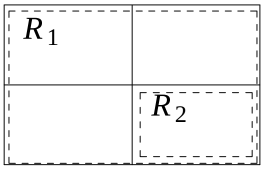


Figure 30 A nested structure

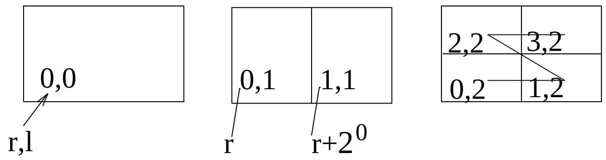
5. Performance Analysis

5.1. Approaches to Performance Comparison

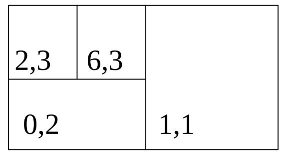
In order to determine a better structure between two given structures, benchmarking of indexes is necessary. Data sets and methods to measure the index performance must be determined. The following parameters on data distribution are likely to affect the performance of an index:

(1) *The number of spatial objects per unit of space*. The distribution of objects in space is irregular, resulting in some areas denser than the others. Objects often overlap in SDS, especially in GIS. However, we should note that certain objects cannot overlap at all, for example, lakes and rivers. The overlap of objects vary, often from 5 to 10 objects [Fra91]. Many indexes involve the process of partitioning of objects into two groups such that each can fit in a page, where each page has an associated data space. A good index should minimize the number of overlapping data spaces. Frank reported that the differences in density can be very substantial and may surpass 1:1000 [Fra81]. For such an overly densed space, indexes based on object duplication are the least ideal structures. Object bounding indexes incur multiple search paths. The proximity preserving mapping techniques will be most suitable, as by mapping non-zero sized objects into points, the space in the higher dimension is not as dense as the original space of lower dimension. However, a more effective distance preserving function is necessary.

(2) *The size of objects*: The sizes of objects are highly dependent on object type. Spatial descriptions of objects are typically extensive, ranging from a few hundred bytes in Land Information System applications to megabytes in natural resource applications [Ab93]. For example, the objects in VLSI and Land Information Systems are more regularly shaped, while the objects in GIS are much larger and irregularly shaped. Many basic spatial operations such as testing the intersection of two polygons are expensive. As a broad indication of the cost of operations, retrieving a polygon from a database of 50000 polygons using an intersection qualification costs in the region of 2 milliseconds on a SUN SPARC-10 machine.



(a) Region number assignment



(b) Non-regular partitioning

Figure 29 The BANG file numbering scheme

When we talk about object sizes, we should constrain ourselves to the application domain, and only consider the relative size. Further, objects in SDS such as GIS are stored in layers, and each layer is typically indexed individually. Large objects mean large covering space, which can result in a denser space. In distributions where there are many large objects, indexes based on object bounding and object duplication are highly affected.

(3) *The Database Size*: Many spatial databases of real-world interest are very large, with sizes ranging from tens of thousands to millions of objects. Nevertheless, as stated in [Fra91], the general rule of "constance of data volume" for an organization applies based on the fact that for smaller application areas, the data collected are more detailed. In a GIS, a billion objects is quite common. In [Goo90], Goodchild reports that a map sheet takes up about 25 mega bytes of memory space and about 300 Gbytes for a county like Los Angeles. With such a huge amount of data and so many indexes that we have, it is not practical to test the indexes on these data sets. A subset of the data, eg. a small area of a county, which is big enough to provide the indexes with reasonable heights is sufficient.

Different distributions can be obtained by varying the parameters. The parameters are inter-related and to see the effect of one over the others, we have to consider all combinations of parameter values. But during result analysis, we can only consider one varying parameter against the cost incurred.

5.2. Comparison Study

In this section, we briefly summarize various performance comparisons that have been conducted in the literature.

To evaluate the effectiveness of the extending methods, we constructed three different indexes which have the same base structure but use different extending methods. Kd-tree variants have been proposed separately in [BaK86], [MHN84] and [OMS87] using object mapping, object duplication and object bounding methods respectively. They are suitable for the evaluation of the effectiveness of the extending methods. In [Ooi90],

Another instance of skewedness is the location of the objects; certain areas are likely to be denser (more objects covering the same area) than others. Different data distributions that follow either one or both of the following skew conditions were used:

(1) The object sizes are skewed, with smaller objects occurring more frequently than larger objects.

(2) The locations of objects are skewed, i.e. the densities are skewed. This is very common in geographic information systems, where a large number of objects occupy certain neighborhoods and a small number of objects are scattered over a relatively

large area.

From the experimental results, the 4d-tree is the least efficient structure. Its nodes store less information than those of the skd-tree, which accounts for a smaller directory size. Intersection search is not supported efficiently because of its inability to prune the search space effectively. It was observed that:

(1) Small objects and low density favor the object duplication approach.

(2) Object bounding approach is in general more efficient and least affected by the type of distributions.

(3) Simple object mapping is the least effective approach.

In [FSR87], analysis of the behavior of the R-tree and R^+-tree on representing one-dimensional intervals of equal length was conducted by transforming the intervals into points in two dimensional space. The same approach was used in [KrS88] for the analysis of PLOP-hashing. Empirical analysis were also conducted in [BKS90, Gre89, KSS89, MHN84 OSM91, SeK90]. In [OSM91], we used the same approach in the previous subsection to compare the performance of the skd-tree and the R-tree. The results indicate that the skd-tree is a more efficient structure than the R-tree [Gut84] with nearly the same storage requirement. The containment search provided by the skd-tree is more efficient than its intersection search and is less sensitive to skewed data. Different distributions such as Gausian and Parcel were used in [BKS90] to generate data for experiments.

In [HoS92], Hoel and Samet conducted a qualitative comparison study on the performance of three spatial indexes, namely the R^*-tree, the R^+-tree, and the PMR quadtree [NeS86, NeS87], on large line segment databases. Tests were conducted on six maps of counties of Maryland, the United States, where each map contains approximately 50,000 line segments. Spatial testing on line segments was conducted. These queries include finding all lines incident at a given point, and at the other endpoint of the line segment of a given point, nearest line segments of a given point, the MBR of line segments that contains a given point and all line segments with a given rectangular window. In their implementation, the execution time of query retrieval is the prime objective, which is sometimes achieved at the expense of a little more expensive storage space. The difference in performance is not very great although, the PMR quadtree has a slight edge over the other two, and the R^+-tree is slightly better than the R^*-tree because of the disjoint decomposition of line segments. The R^+-tree required considerably more space than the other two structures. However, the study did not result in claims of convincing superiority for any of the tested three indexes. This could be due to the use of line segments, which are much simpler than non-zero sized and irregularly shaped objects.

In [Ooi90], the efficiency of three extending methods was studied using a family of kd-trees, namely skd-trees [OMS87], Matsuyama kd-tree [MHN84], and the 4d-tree [BaK86]. Databases of 12000 objects were generated with different distribution of object sizes and object locations. The average data density used is 3. However, for very skewed object placements, the data density of certain locations could be very high. The study shows that the Matsuyama kd-tree which adopts the non-overlapping native space indexing approach performs efficiently in terms of page accesses for small objects. As the object sizes become bigger, its performance degrades. The 4d-tree is n

In [PTS95], the topological relationships of *meet, overlap, inside, covered-by, covers, contains, and disjoint* between MBRs were studied. The efficiency of the R-tree, R_+-tree, and R_*-tree were then studied using three databases of 10,000 objects, with different sizes of MBRs, and 100 queries. For small MBRS (less than 0.02% of the map area) and medium MBRs (less than 0.1% of the map area), R_*-trees and R_+-trees outperform the R-tree, with the R_+ slightly more efficient than the R_*-tree. However, for large MBRs (less than 0.5% of the map area), the R_+ becomes less efficent than the other two due to the additional level caused by duplications. As experienced in [Gre89, PTS95], the R_+ does not work for high data density.

References and Bibliography

[AbS83] D. J. Abel, J. L. Smith: A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Int. Journal of Comp. Vision, Graphics, and Image Processing 24*, 1, 1-13 (1983).

[AbS84] D. J. Abel, J. L. Smith: A data structure and query algorithm for a database of areal entities. *Aust. Comp.J. 16*, 4, 147-154 (1984).

[AbS86] D. J. Abel, J. L. Smith: A relational GIS database accommodating independent partitionings of the region. *Proc. 2nd Int. Symp. on Spatial Data Handling*, Seattle, WA, 213-224 (1986).

[AbS87] D. J. Abel, J. L. Smith: A kernel-shell approach to an extended relational spatial database management system. Unpublished paper, CSIRO (1987).

[Abe84] D. J. Abel: A B+ structure for large quadtrees. *Int. Journal of Comp. Vision, Graphics, and Image Processing 27*, 1, 19-31 (1984).

[Abe86] D. J. Abel: Bit-interleaved keys as the basis for spatial access in a front-end spatial database management system. In: B. Diaz, S. Bell, (eds): *Spatial Data Processing Using Tesseral Methods*, Natural Environment Research Council, 163-177 (1986).

[Abe89] D. J. Abel: SIRO-DBMS: a database tool-kit for geographical information systems. *Int. Journal on Geographical Inf. Sys. 3*, 2, 103-116 (1989).

[Abe90] D. J. Abel: A model for data set management in large spatial information systems, *Int'l. Journal on Geographical Information Systems 3*, 291-302 (1990).

[AHU74] A. V. Aho, J. E. Hopcroft, J. D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA (1974).

[ArS91] W. G. Aref, H. Samet: Optimization strategies for spatial query processing. *Proc. 17th Int. Conf. on Very Large Data Bases*, 81-90 (1991).

[BaK86] J. Banerjee, W. Kim: Supporting VLSI geometry operations in a database system. *IEEE Proc. Data Engineering Conf.*, Los Angeles, CA, 409-415 (1986).

[BaM72] R. Bayer, E. McCreight: Organization and maintenance of large ordered indices. *Acta Informatica 1*, 3, 173-189 (1972).

[BER85a] D. A. Beckley, M. W. Evens, V. K. Raman: Empirical comparison of associative file structures. *Proc. Int. Conf. on Foundations of Data Organisation*, Kyoto, Japan, 315-319 (1985).

[BER85b] D. A. Beckley, M. W. Evens, V. K. Raman: An experiment with balanced and unbalanced K-D trees for associative retrieval. *Proc. IEEE COMPSAC - Comp. Software & Applications Conf.*, Chicago, IL, 256-262 (1985).

[BER85c] D. A. Beckley, M. W. Evens, V. K. Raman: Multikey retrieval from K-D trees and quad trees. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Austin, Texas, 291-301 (1985).

[BKS90] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger: The R^*-tree: An efficient and robust access method for points and rectangles. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, 322-331, (1990).

[Ben75] J. L. Bentley: Multidimensional binary search trees used for associative searching. *Comm. ACM 18*, 9, 509-517 (1975).

[BeF79] J. L. Bentley, J. H. Friedman: Data structures for range searching. *ACM Comp. Surveys 11*, 4, 397-409 (1979).

[Ben79a] J. L. Bentley: Multidimensional binary search trees in database applications. *IEEE Trans. on Software Eng. SE-5*, 4, 333-340 (1979).

[Ben79b] J. L. Bentley: Decomposable searching problems. *Information Processing Letters 8*, 5, 244-251 (1979).

[BlM90] H. Blanken, A. Ijbema, P. Meek, B v d Akker: The generalized grid file: Description and performance aspects. *IEEE Proc. 6th Int. Conf. on Data Engineering*, L.A. 380-388 (1990).

[Bra84] K. Bratbergsengen: Hashing methods and relational algebra operations. *Proc. 10th Int. Conf. on Very Large Data Bases*, Singapore, 323-333 (1984).

[BKS93a] T. Brinkhoff, H.-P. Kriegel, B. Seeger: Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. *Proc. Int'l. Conf. on Data Engineering*, 40-49 (1993).

[BKS93b] T. Brinkhoff, H.-P. Kriegel, B. Seeger: Efficient processing of spatial joins using R-trees. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 237-246 (1993).

[BKS94] T. Brinkhoff, H.-P. Kriegel, R. Schneider, B. Seeger: Multi-step processing of spatial joins. *ACM SIGMOD Int. Conference on Management of Data*, 197-208 (1994).

[Bur83] W. A. Burkhard: Interpolation-based index maintenance. *BIT 23*, 274-294 (1983).

[CeS82] F. Cesarini, G. Soda: Binary trees paging, *Information Systems 7*, 4, 337-344 (1982).

[COL92] C. Y. Chan, B. C. Ooi, H. Lu: Extensible buffer management of indexes. *Proc. Int. Conf. on Very Large Data Bases*, 444-455 (1992).

[ChF79] J. M. Chang, K. S. Fu: Extended K-D tree database organization: A dynamic multi-attribute clustering method. *Proc. IEEE COMPSAC - Comp. Software & Applications Conf.*, Chicago, IL, 39-43 (1979).

[CHH92] X. Cheng, H. Lu, G. E. Hedrick: Searching spatial objects with index by dimensional projections. *Proc. ACM/SIGAPP Symp. on Applied Computing*, Vol. 1, Kansas City, 217-223 (1992).

[CCK81] M. Chock, A.F. Cardenas, A. Klinger: Manipulating data structures in pictorial information systems. *IEEE Computer 14*, 11, 43-52 (1981).

[Com79] D. Comer: The ubiquitous B-Tree. *ACM Comp. Surveys 11*, 2, 121-137 (1979).

[DaS85] S. P. Dandamudi, P. G. Sorenson: An empirical performance comparison of some variations of the k-d tree and BD tree. *Int'l. Journal of Computer and Information Sciences 14*, 3, 135-159 (1985).

[DaH85] W. A. Davis, C. H. Hwang: File organization schemes for geometric data. Tech. Rep. 85-14, Univ. of Alberta, Edmonton, Canada (1985).

[EaZ82] C. M. Eastman, M. Zemankova: Partially specified nearest neighbour using kd Trees. *Information Processing Letters 15*, 2, 53-56 (1982).

[DrS93] G. Droege, H.-J. Schek: Query-adaptive data space partitioning using variable-size storage clusters. *Proc. 3Rd Int'l. Symposium on Large Spatial Databases,* Lecture Notes in Computer Science #692, 337-356, (1993).

[FNP79] R. Fagin, J. Nievergelt, N. Pippenger, H. R. Strong: Extendible hashing - A fast access method for dynamic files. *ACM Trans. on database sys. 4,* 3, 315-344 (1979).

[FaR91] C. Faloutsos, Y. Rong: DOT: A spatial access method using fractals. *Proc. Int. Conf. on Data Eng.*, Kobe, Japan, 152-159 (1991).

[FSR87] C. Faloutsos, T. Sellis, N. Roussopoulos: Analysis of object oriented spatial access methods. *Proc. ACM SIGMOD Int. Conf. on management of data*, San Francisco, California, 426-439 (1987).

[FiB74] R. A. Finkel, J. L. Bentley: Quad Trees: A data structure for retrieval on composite keys. *Acta informatica 4,* 1-9 (1974).

[Fra81] A. Frank: Applications of DBMS to land information systems. *Proc. Int. Conf. on Very Large Data Bases*, 448-453 (1981).

[Fra82] A. Frank: Mapquery: Database query language for retrieval of geometric data and its graphical representation. *ACM SIG Computer Graphics 16*, 3, 199-207 (1982).

[Fra91] A. Frank: Properties of geographical data: requirements for spatial access methods. *Proc. 2nd Int. Symp. on Large Spatial Database*, Lecture Notes in Computer Science #525, Springer-Verlag, 225-236 (1991).

[Fre87] M. Freeston: The BANG file: A new kind of grid file. *Proc. ACM SIGMOD Int. Conf. on management of data*, San Francisco, California, 260-269 (1987).

[Fre89a] M. Freeston: Advances in the design of BANG file. *3rd. Int. Conf. on Foundations of Data Organization and Algorithms*, Paris, Lectures Notes in Computer Science #367, Springer-Verlag (1989).

[Fre89b] M. Freeston: A well behaved file structure for the storage of spatial objects. *First Int. Symp. on Large Spatial Databases,* California, Lecture Notes in Computer Sc. #409, (1989).

[FBF77] J. H. Friedman, J. L. Bentley, R. A. Finkel: An algorithm for finding best matches in logarithmic expected time. *ACM Trans. on Math. Software 3*, 3, 209-226 (1987).

[GaR88] M. N. Gahegan, S. A. Roberts: Intelligent object-oriented geographic information system. Unpublished paper, Dept. of Computer Studies, Uni. of Leeds, Leeds, England (1988).

[Goo90] M. F. Goodchild: Tiling a large geographical database. *Proc. Int. Symp. on Large Spatial Databases*, (1989).

[Gue94] R. H. Gueting: An introduction to spatial database systems. *The VLDB Journal 3*, 4, 357-400, (1994).

[Gun88] O. Gunther: *Efficient structures for geometric data management*. Lecture Notes in Computer Science 337, Springer-Verlag (1988).

[Gun89] O. Gunther: The design of the cell tree: an object-oriented index structure for geometric databases. *IEEE 5th Int. Data Engineering Conf.*, 598-605 (1989).

[Gun93] O. Gunther: Efficient computations of spatial joins. *Proc. IEEE Int. Conf. on Data Engineering*, 50-59 (1993).

[GuB90] O. Gunther, A. Buchmann: Research Issues in Spatial Databases. *ACM SIGMOD Record 19,* 4, 61-68 (1990).

[GuN91] O. Gunther, H. Noltemeier: Spatial database indices for large extended objects. *Proc. Int. Conf. on Data Eng.*, Kobe, Japan, 520 -526 (1991).

[Gut84] A. Guttman: R-trees: A dynamic index structure for spatial searching. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Boston, MA, 47-57 (1984).

[Gre89] D. Greene: An implementation and performance analysis of spatial data access methods. *Proc. 5th Int. Conf. on Data Engineering*, 606 - 615 (1989).

[Har80] R. M. Haralick: A spatial data structure for geographic information. In: H. Freeman, G. G. Pieroni (eds): *Map Data Processing*, Maratea, Italy, Academic Press, New York, 63-100 (1986).

[HSW89a] A. Henrich, H.-W. Six, P. Widmayer: Paging binary trees with external balancing. *Proc. Int. Workshop on Graphtheoritic Concepts in Comp. Sc.*, Springer Verlag, Lecture Notes in Computer Sc., (1989).

[HSW89b] A. Henrich, H.-W. Six, P. Widmayer: The LSD tree: spatial access to multidimensional point and non-point objects. *Proc. Int. Conf. on Very Large Data Bases*, 45-53 (1989).

[Hil91] D. Hilbert: Uber die steitige Abbildung einer Linie auf ein Flachenstuck. *Math. Ann. 38* (1891).

[HiN83] K. Hinrichs, J. Nievergelt: The grid file: A data structure designed to support proximity queries on spatial objects. *Proc. Int. Workshop on*

Graphtheoritic Concepts in Comp. Science. Trauner-Verlag, 100-113 (1983).

[Hin85] K. Hinrichs: Implementation of the grid file: Design concepts and experience. *BIT 25*, 569-592 (1985).

[HoS92] E. G. Hoel, H. Samet: A qualitative comparison study of data structures for large line segment databases. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 205-214 (1992).

[HSW88a] A. Hutflesz, H.-W. Six, P. Widmayer: The twin grid file: a nearly space optimal index structure. *Proc. Int. Conf. on Extending Database Technology*, (1988).

[HSW88b] A. Hutflesz, H.-W. Six, P. Widmayer: Twin grid files: space optimizing access schemes. *ACM SIGMOD Int. Conf. on Management of Data*, Illinois, 183-190 (1988).

[HSW88c] A. Hutflesz, H.-W. Six, P. Widmayer: Globally order preserving multidimensional linear hashing. *Proc. IEEE 4th Int. Conf. on Data Eng.*, 572-579 (1988).

[HSW90] A. Hutflesz, H.-W. Six, P. Widmayer: The R-file: An efficient access structure for proximity queries. *Proc. IEEE 6th Int. Conf. on Data Engineering*, 372-379 (1990).

[Jag90a]

H.V. Jagadish: Spatial search with polyhedra. *Proc. IEEE 6th Int. Conf. on Data Engineering*, 311-319 (1990).

[Jag90b] H.V. Jagadish: On indexing line segments. *Proc. 16th Int. Conf. on Very Large Data Bases*, 614-625 (1990).

[KaF92] I. Kamel, C. Faloutsos: Parallel R-trees. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 195-204 (1992).

[KeW87] A. Kemper, M. Wallrath: An analysis of geometric modeling in database systems. *ACM Comp. Surveys 19*, 1, 47-91 (1987).

[KFC90] S. Khoshafian, M. J. Franklin, M. J. Carey: Storage management for persistent complex objects. *Inform. Sys. 15*, 3, 303-320 (1990)

[Kli71] A. Klinger: Patterns and search statistics. In: J. S. Rustagi (ed): *Optimizing Methods in Statistics*, Academic Press, New York, 307-337 (1971).

[Knu73] D. E. Knuth: *Fundamental Algorithms, 2nd Edition*. The art of computer programming, vol. 1, Reading, MA: Addison-Wesley (1973).

[Kri84] H. P. Kriegel: Performance comparison of index structures for multi-key retrieval. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Boston, MA, 186-196 (1984).

[KSS89] H. Kriegel, M. Schiwietz, R. Scneider, B. Seeger: Performance comparison of point and spatial access methods. *First Symp. on Large Spatial Databases*, Lecture Notes in Computer Sc. #409, (1989).

[KrS86] H. Kriegel, B. Seeger: Multidimensional order preserving linear hashing with partial expansion. *Proc. Int. Conf. on Database Theory*, Springer-Verlag, New York, 203-220 (1986).

[KrS88] H. Kriegel, B. Seeger: PLOP-Hashing: A grid file without directory. *Proc. IEEE 4th Int. Conf. on Data Engineering*, 369-376 (1988).

[KBS91] H. Kriegel, T. Brinkhoff, R. Schneider. The combination of spatial access methods and computational geometry in geographic database systems. *Proc. 2nd Int. Symp. on Large Spatial Database*, Lecture Notes in Computer Science #525, Springer-Verlag, 5-22 (1991).

[KHS91] H. Kriegel, H. Horn, M. Schiwietz: The performance of object decomposition techniques for spatial query processing. *Proc. 2nd Int. Symp. on Large Spatial Database*, Lecture Notes in Computer Science #525, Springer-Verlag, 257-275 (1991).

[Lar78] P. Larson: Dynamic hashing. *BIT 13*, 184-201 (1978).

[LeW77] D. T. Lee, C. K. Wong: Worst-Case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica 9*, 1, 23-29 (1977).

[LeB92] J.-T. Lee, G. Belford: An efficient object-oriented algorithm for spatial searching, insertion and deletion. *Proc. IEEE 8th Int. Conf. on Data Engineering*, 40-47 (1992).

[LiL91] K. Li, R. Laurini: The spatial locality and a spatial indexing method by dynamic clustering. *Proc. 2nd Int. Symp. on Large Spatial Database*, Lecture Notes in Computer Science #525, Springer-Verlag, 207-224 (1991).

[LiC79] B. S. Lin, S. K. Chang: Picture algebra for interface with pictorial database systems. *Proc. IEEE COMPSAC - Comp. Software & Applications Conf.*, 525-530 (1979).

[LJF94] K.I. Lin, H.V. Jagadish, C. Faloutsos: The TV-tree: an index structure for high-dimensional data. *The VLDB Journal 3*, 4, 517-542, (1994).

[LoR94] M.-L. Lo, C.V. Ravishankar: Spatial joins using seeded trees. *Proc. ACM SIGMOD Int. Conference on Management of Data*, 209-220 (1994).

[LoR95] M.-L. Lo, C.V. Ravishankar: Generating seeded trees from data sets. *Proc. 4th Int. Symposium on Large Spatial Databases*, Lecture Notes in Computer Science #951, Springer-Verlag, 328-347, (1995).

[Lom91] D. Lomet: Grow and post index trees: role, techniques and future potential. *Proc. 2nd Int. Symp. on Large Spatial Database*, Lecture Notes in Computer Science #525, Springer-Verlag, 183-206 (1991).

[Lom92] D. Lomet: A review of recent work on multi-attribute access methods. *ACM SIGMOD Record 21*, 3, 56-63 (1992).

[LoS89] D. Lomet, B. Salzberg: The hB-tree: a robust multi-attribute search indexing method. *Proc. IEEE 5th Int. Conf. on Data Engineering*, (1989).

[LoS90] D. Lomet, B. Salzberg: The hB-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans. on Database Sys. 15*, 4, (1990).

[LuH92] W. Lu, J. Han: Distance-associated join indices for spatial range search. *Proc. 8th IEEE. Int. Conf. on Data Engineering*, 284-292 (1992).

[LuO93] H. Lu, B. C. Ooi: Spatial indexing: Past and future. *IEEE Bulletin on Data Engineering*, (to appear) (1993).

[LOD91] H. Lu, B. C. Ooi, A. D'Souza, C. C. Low: Storage management in geographic information systems. *Int. Symp. on Large Spatial Database Systems* Lecture Notes in Computer Science #525, 451-470 (1991).

[Man88] M. Mantyla: *An introduction to solid modeling*. Computer Science Press, Rockville, MD (1988).

[Mar82] J. J. Martin: Organisation of geographic data with quad trees and least square approximation. *Proc. Conf. on Pattern Recognition and Image Processing*, Las Vegas, Nevada, 458-463 (1982).

[MHN84] T. Matsuyama, L.V. Hao, M. Nagao: A file organization for geographic information systems based on spatial proximity. *Int. Journal Comp. Vision, Graphics, and Image Processing 26*, 3, 303-318 (1984).

[McK84] D. M. McKeown: Digital cartography and photo interpretation from a data base viewpoint. In: G. Gardarin and E. Gelenbe (eds): *New Applications of Data Bases*, Academic Press, London, 19-42 (1984).

[McK83] D. M. McKeown Jr: MAPS: the organization of a spatial data base system using imagery, terrain and map data. Tech. Rep. CMU-Comp. Science-83-136, Comp. Sc. Dept., Carnegie-Mellon University (1983).

[NaW79] G. Nagy, S. Wagle: Geographic data processing. *ACM Comp. Surveys 11*, 2, 139-181 (1979).

[NAO88] Y. Nakamura, A. Abe, Y. Ohsawa, M. Sakauchi: MD-tree: A balanced hierarchical data structure for multidimensional data with highly efficient dynamic characteristics. *Proc. IEEE Int. Conf. on Pattern Recognition*, 375-378 (1988).

[NeS86] R. C. Nelson, H. Samet: A consistent hierarchical representation for vector data. *Computer Graphics 20*, 4, 197-206 (1986).

[NeS87] R. C. Nelson, H. Samet: A population analysis for hierarchical data structures. *Proc. ACM Int. Conf. on Management of Data*, 270-277 (1987).

[Nie89] J. Nievergelt: 7 ± 2 criteria for assessing and comparing spatial data structures. *First Symp. on Large Spatial Databases*, California, Lecture Notes in Computer Sc. #409, 3-27 (1989).

[Nie95] J. Nievergelt: Private communication, Singapore, April (1995).

[NiH85] J. Nievergelt, K. Hinrichs: Storage and access structures for geometric data bases. *Int. Conf. on Foundations of Data Organization*, Kyoto, Japan, 335-345 (1985).

[NHS81] J. Nievergelt, H. Hinterberger and K.C. Sevcik: The Grid File: An adaptable, symmetric multikey file structure. In Trends in Information Processing Systems, i*Proc. 3rd ECI Conf.*, in A. Duijvestijn and P. Lockemann (eds.), Lecture Notes in Computer Science #123, 236-251, Springer Verlag (1981).

[NHS84] J. Nievergelt, H. Hinterberger, K. C. Sevcik: The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. on Database Sys. 9*, 1, 38-71 (1984).

[NgK93] V. Ng, T. Kameda: Concurrent access to R-trees. *Proc. the 3rd Int'l. Symposium on Large Spatial Databases*, Lecturer Notes in Computer Science 692, Springer-Verlag, 142-161 (1993).

[OhS83] Y. Ohsawa, M. Sakauchi: The BD-tree — a new n-dimensional data structure with highly efficient dynamic characteristics. *Proc. IFIP Congress*, North-Holland, 539-544 (1983).

[OhS90] Y. Ohsawa, M. Sakauchi: A new tree type data structure with homogeneous nodes suitable for a very large spatial database. *Proc. IEEE 6th Int. Conf. on Data Engineering*, 296-303 (1990).

[Ooi88] B. C. Ooi: *Efficient Query Processing in Geographic Information Systems*. Ph.D. Thesis, Monash University, Australia (1988). (a revised version appears in: Lecture Notes in Computer Science #471, Springer-Verlag, 1990).

[OMS87] B. C. Ooi, K. J. McDonell, R. Sacks-Davis: Spatial kd-tree: An indexing mechanism for spatial databases. *Proc. IEEE Int. Comp. Software & Applications Conf.*, Japan (1987).

[OSM89] B. C. Ooi, R. Sacks-Davis, K. J. McDonell: Extending a DBMS for geographic applications. *Proc. IEEE 5th Int. Data Engineering Conf.*, 590-597 (1989).

[OSM91] B. C. Ooi, R. Sacks-Davis, K. J. McDonell: Spatial indexing by binary decomposition and spatial bounding. *Information Sys. Journal 16*, 2, 211-237 (1991).

[Oos90] P. v. Oosterom: *Reactive Data Structures for Geographic Information Systems*, Ph.D. thesis, Leiden University (1990).

[OoB89] P. v. Oosterom, J. v. d. Bos: An object-oriented approach to the design of geographic information systems. *Computers and Graphics 13*, 4, (1989).

[Ore82] J. A. Orenstein: Multidimensional tries for associative searching. *Information Processing Letters 14*, 4, 150-157 (1982).

[Ore86] J. A. Orenstein: Spatial query processing in an object-oriented database system. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Washington, D.C., 326-336 (1986).

[Ore89] J. A. Orenstein: Redundancy in spatial databases. *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 294-305 (1989).

[Ore90] J. A. Orenstein: A comparison of spatial query processing techniques for native and parameter spaces. *Proc. ACM Int. Conf. on Management of Data*, 343-352 (1990).

[OrM88] J. A. Orenstein, F. A. Manola: PROBE spatial data modeling and query processing in an image database application. *IEEE Trans. on Software Eng. 14*, 5, 611-629 (1988).

[OrM84] J. A. Orenstein, T. H. Merrett: A class of data structures for associative searching. Proc. *ACM PODS Conf.*, 181-190 (1984).

[Oto85] E. J. Otoo: A multidimensional digital hashing scheme for files with composite keys. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 214-229 (1985).

[OuS81] M. Ouksel, P. Scheuermann: Multidimensional B-trees: Analysis of dynamic behavior. *BIT 21*, 401-418 (1981).

[OuM92] M. Ouksel, O. Mayer: A robust and efficient data structure: The nested interpolation-based grid file. *Acta Informatica 29*, (1992).

[Ous84] J. K. Ousterhout: Corner stitching: A data structuring technique for VLSI layout tools. *IEEE Tans. on Comp. Aided Design CAD-3*, 1, 87-100 (1984).

[OvL82] M. H. Overmars, J. V. Leeuwen: Dynamic multi-dimensional data structures based on quad- and KD- trees. *Acta Information 17*, 267-285 (1982).

[PST93] B.-U. Pagel, H.-W. Six, H. Toben: The transformation technique for spatial objects revisited. *Proc. The 3rd In'tl. Symposium on Large Spatial Databases*, Lecture Notes in Computer Science 692, Springer-Verlag, 73-88 (1993).

[PTS95] D. Papadias, Y. Theodoridis, T. Sellis, M. J. Egenhofer: Topological relations in the world of minimum bounding rectangles: a study with R-trees. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 92-103 (1995).

[PrS85] F. P. Preparata, M. I. Shamos: *Computational Geometry: an introduction*, Springer-Verlag, (1985).

[Rea94] W. W. Read: On the average storage utilization of hB-trees. *Proc. 5th Australasian Database Conference*, 112-123 (1994).

[Req80] A. A. G. Requicha: Representations for solids: theory, methods, and systems. *ACM Computing Survey 12*, 4, 437-464 (1980).

[ReV83] A. A. G. Requicha, H. B. Voelcker: Solid modeling: current status and research directions. *IEEE Comp. Graphics App.*, 25-37 (1983).

[Rob81] J. T. Robinson: The K-D-B-tree: A search structure for large multidimensional dynamic indexes. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 10-18 (1981).

[Rot91] D. Rotem: Spatial join indices. *Proc. IEEE 7th Int. Conf. on Data Eng.*, 500-509 (1991).

[Ros85] J. B. Rosenberg: Geographical data structures compared: A study of data structures supporting region queries, *IEEE Trans. on Comp. Aided Design CAD-4*, 1, 53-67 (1985).

[RoL84] N. Roussopoulos, D. Leifker: An introduction to PSQL: A pictorial structured query language. *Proc. IEEE Workshop on Visual Language*, 77-87 (1984).

[RoL85] N. Roussopoulos, D. Leifker: Direct spatial search on pictorial databases using packed R-trees. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 17-31 (1985).

[RFS88] N. Roussopoulos, C. Faloutsos, T.K. Sellis: An efficient pictorial database system for PSQL. *IEEE Trans. on Software Eng. 14*, 5, 639-650 (1988).

[SMO87] R. Sacks-Davis, K. J. McDonell, B. C. Ooi: GEOQL - A query language for geographic information systems. Tech. Rep. 87/2, Dept. of Computer Sc. R.M.I.T. Melbourne.

[Sam80] H. Samet: The Deletion in two-dimensional quad-trees. *ACM Comm. of ACM 23*, 12, 703-710 (1980).

[Sam84] H. Samet: The quadtree and related hierarchical data structures. *ACM Comp. Surveys 16*, 187-260 (1984).

[Sam86] H. Samet: *Bibliography on Quadtrees and Hierarchical Data Structures*. Computer Sc. Dept., University of Maryland, Maryland (1986).

[Sam88] H. Samet: Hierarchical representations of collections of small rectangles. *ACM Comp. Surveys* 20, 4, 271-309 (1988).

[SaW85] H. Samet, R. E. Webber: Storing a collection of polygons using quadtrees. *ACM Trans. on Graphics 4*, 3, 182-222 (1985).

[ScO82] P. Scheuermann, M. Ouksel: Multidimensional B-trees for associative searching in database systems. *Information Systems 7*, 2, 123-137 (1982).

[ScK93] M. Schiwietz, H.-P. Kriegel: Query processing of spatial objects: complexity versus redundancy. *Proc. The 3rd Int'l. Symposium on Large Spatial Databases*, Lecture Notes in Computer Science 692, Springer-Verlag, 377-396 (1993).

[See91] B. Seeger: Performance comparison of segment access methods implemented on top of the buddy-tree. *Proc. 2nd Int. Symp. on Large Spatial Database*, Lecture Notes in Computer Science #525, Springer-Verlag, 277-296 (1991).

[SeK88] B. Seeger, H. Kriegel: Techniques for design and implementation of efficient spatial access methods. *Proc. 14th Int. Conf. on Very Large Data Bases*, L.A., California, 360-371 (1988).

[SeK90] B. Seeger, H. Kriegel: The buddy-tree: An efficient and robust access method for spatial data base systems. *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, 590 - 601 (1990).

[SRF87] T. Sellis, N. Roussopoulos, C. Faloutsos: The R^+-tree: A dynamic index for multi-dimensional objects. *Proc. 13th Int. Conf. Very Large Data Bases*, Brighton, England, 507-518 (1987).

[Sha86] C. A. Shaffer: Application of alternative quadtree representation. MCS-83-02118, Ph.D. Thesis, Center for Automation Research, Uni. of Maryland, College Park (1986).

[ShB78] M. I. Shamos, J. L. Bentley: Optimal algorithm for structuring geographic data. *Proc. 1st Int. Advanced Study Symp. on Topological Data Structure for Geographic Inf. Sys. 6*, Dedham, Mass., Harvard Univ. Lab for Computer Graphics and Spatial Analysis (1978).

[ShH78] L. G. Shapiro, R. M. Haralick: A general spatial structure. *Proc. Conf. on Pattern Recognition and Image Processing*, Chicago, IL, 238-249 (1978).

[ShR85] K. D. Sharma, R. Rani: Choosing optimal branching factors for k-d-b trees. *Information Systems 10*, 1, 127-134 (1985).

[SiW88] H. Six, P. Widmayer: Spatial searching in geometric databases. *IEEE 4th Int. Conf. on Data Engineering*, L. A., California, 496-503 (1988).

[Tam82a] M. Tamminen: Efficient spatial access to a data base. *Proc. SIGMOD Int. Conf. on the Management for Data*, ACM, 200-206 (1982).

[Tam82b] M. Tamminen: The extendible cell method for closest points problems. *BIT 22*, 27-41 (1982).

[Tam83] M. Tamminen: Performance analysis fo cell based geometric file organisations. *Int. Journal Comp. Vision, Graphics, and Image Processing 24*, 160-181 (1983).

[TaS82] M. Tamminen, R. Sulonen: The EXCELL method for efficient geometric access to data. *Proc. ACM IEEE 19th Design Automation Conf.*, Las Vegas, Nevada, 345-351 (1982).

[ThT94] Y. Theodoridis, T. Sellis: Optimization issues in R-tree construction. *Proc. Int'l. Workshop on Advanced Research in Geographical Information Systems*, Lecture Notes in Computer Science #884, 270-273 (1994).

[TKN80] T. Tsurutani, Y. Kasahara, M. Naniwada: ATLAS: A geographic database system - data structure and language design for geographic information. *Comp. Graphics*, ACM, New York, 71-77 (1980).

[Val87] P. Valduriez: Join indices. *ACM Trans. on Database Systems 12*, 2, 218-246 (1987).

[WhK85] K. Whang, R. Krishnamurthy: Multilevel grid files. Report RC 11516, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1985).