SPATIAL DATA STRUCTURES: CONCEPTS AND DESIGN CHOICES

J Nievergelt

Rapporteur: Michael Elphick

Spatial data structures: concepts and design choices

Jürg Nievergelt and Peter Widmayer
ETH Zurich

Abstract

Spatial data structures, once considered mere variations of structures designed for conventional (non-spatial) data, increasingly evolve along a different track. The reason for this lies in the rich structure of Euclidean space. Even without the presence of objects, regions of space admit a multitude of operations and relations that are relevant for efficient query processing, such as: distance, containment, intersection, touching, in front of. This fact suggests that the main task of a spatial data structure is to organize the embedding space in a systematic manner, rather than the content, i.e. the transient set of objects present at any moment. This is particularly true when objects move or are subject to other transformations, as is the case in applications such as computer-aided design (CAD).

A second difference between spatial and non-spatial data lies in the query population. Rather than retrieving objects based on their inherent characteristics, the majority of spatial queries are based on proximity of the objects to be retrieved to query regions of the most varied shape (rays, boxes, arbitrary polyhedra). This implies that retrieval begins with a coarse filter that retrieves a cluster of neighboring objects, followed by a fine filter that analyzes their geometric properties. Thus, proximity in space should be reflected in contiguous storage to the extent possible.

This survey is concerned with spatial data structures suitable for external storage devices. This implies a conceptual and technical simplicity that excludes a few sophisticated list structures designed for central memory. We explain concepts, and their intuitive justification, in the context of their historical development. We emphasize common sense principles more than detailed techniques; and we illustrate basic ideas by presenting a sample of

spatial data structures selected by virtue of their conceptual simplicity. Thus, we hope to provide a user's guide to an applications programmer confronted with a bewildering multitude of structures described in the research literature, many of which differ only in detail: "If you ask the right questions, common sense will provide answers".

Keywords and phrases: Data management systems, data bases, data structures, geometric and solid modeling, computational geometry, spatial data.

1 Goals and structure of this survey

The growing importance of graphic user interfaces and of applications such as computer-aided design and geo-information systems has confronted many applications programmers with a challenging new task: Processing large amounts of spatial data, off disk, correctly and efficiently. The task is daunting, as spatial data poses distinctly novel problems as compared to traditional "business data", in particular the following two:

1. Access is primarily via proximity relations to other objects that populate Euclidean space, rather than via inherent properties of objects, such as attribute values. A typical query is of the form "retrieve all objects that intersect a given region of space".

2. Guaranteed correctness in processing spatial objects has proven to be a thorny problem requiring a systematic analysis of degenerate configurations. Moreover, efficiency depends on the interplay between two techniques: one for representing an object (independently of its location in space), the other for storing the entire collection of objects in relation to their position in space.

We aim to provide a guide through the bewildering multitude of concepts, techniques and choices a programmer faces when designing a data structure for managing spatial data. We simplify this task by separating the question of how an object is represented for internal processing (an issue in computational geometry that we shall not address), from the question of how a large collection of simple objects embedded in space are managed on disk (the crucial issue of spatial data management systems). Although object representation and external data structures may be intertwined, for example in image processing, they are treated separately in many important applications. In geographic information systems, for example, objects of complex shape are routinely approximated or bounded by simple containers for retrieval purposes.

The tutorial approach we have chosen for this survey begins with the historical development of data structures in general, a trend from which spatial data in particular emerged as a separate discipline relatively late. In spite of the profound differences between spatial data on the one hand, and conventional data typical of the business applications that shaped the development

of database technology on the other hand, the community of database researchers persisted in forcing "non-standard (including spatial) data" into a mold that had proven effective for other applications. It took a long time to realize that data structures inherited or adapted from single-key access had to be reconsidered, and often abandoned. Thus a brief sketch of the development of data structures in section 2 serves the double purpose of reviewing the basic notions of data structures, and of identifying key differences between spatial and other data.

A second characteristic of our tutorial approach is to start from about a dozen concepts or features that appear to cover most of the many spatial data structures described in the literature. This approach assigns to every data structure a "profile" that facilitates assessment of its strong and weak points, and comparison with others. Thus we avoid enumerating the many spatial data structures described in the literature, which often differ only in details, and instead focus on the building blocks from which a programmer can assemble his own data structure tailored to the specific requirements of a given task.

Thus, section 3 presents important concepts needed to discuss spatial data structures at an intuitive level. This includes a list of basic questions to be raised and considerations to be aware of before a programmer decides on data representation and storage technique. The latter have a pervasive influence on the complexity and efficiency of any application that processes spatial data.

Having assembled an arsenal of concepts and terminology, section 4 presents a concise model that captures the essence of the majority of spatial data structures and the way they process queries. By separating three key aspects: the organization of the embedding space into a collection of cells, the organization of objects into cell populations, and the internal representation of an object, we arrive at a three-step query processing model that serves as a skeleton for understanding, assessing and comparing most spatial data structures.

So far we have merely mentioned a few examples of specific data structures, out of several dozen known today. Sections 5 and 6 continue by systematically listing the major building blocks that go into the design of a spatial data structure, and providing examples of structures for possible design choices. Section 5 treats points, the simplest kind of spatial object, whereas Section 6 discusses extended objects. By following an approach "from gen-

eral concepts to specific examples", we hope to help a reader to program an appropriate data structure based on half a dozen design choices, rather than by scanning a vast literature that includes hundreds of research papers.

The concluding section 7 attempts to summarize the survey with a simple point of view: Design and choice of spatial data structures, today, is a matter of common sense more than of technical wizardry. Don't let hundreds of research papers prevent you from seeing the forest because of all the trees. With a clear understanding of a dozen fundamental concepts characteristic of spatial data, you can choose a data structure suited to your application.

2 Data structures old and new, and the forces that shaped them

2.1 Prehistory, logical vs. physical structure

The discipline of data structures, as a systematic body of knowledge, is truly a creation of computer science. The question of how to organize data was a lot simpler to answer in the days before the existence of computers: The organization had to be simple, because there was no automatic device capable of processing intricately structured data, and there is no human being with enough patience to do it. Consider two examples.

1. Manual files and catalogs, as used in business offices and libraries, exhibit several distinct organizing principles, such as sequential and hierarchical order and cross-references. From today's point of view, however, manual files are not well-defined data structures. For good reasons, people did not rigorously define those aspects that we consider essential when characterizing a data structure: what constraints are imposed on the data, both on the structure and its content; what operations the data structure must support; what constraints these operations must satisfy. As a consequence, searching and updating a manual file is not typically a process that can be automated: It requires common sense, and perhaps even expert training, as is the case for a library catalog.

2. In manual computing (with pencil and paper or a nonprogrammable calculator) the algorithm is the focus of attention, not the data struc-

ture. Most frequently, the person computing writes data (input, intermediate results, output) in any convenient place within his field of vision, hoping to find them again when he needs them. Occasionally, to facilitate highly repetitive computations (such as income tax declarations), someone designs a form to prompt the user, one operation at a time, to write each data item into a specific field. Such a form specifies both an algorithm and a data structure with considerable formality, but is necessarily special purpose.

Edge-notched cards are perhaps the most sophisticated data structures ever designed for manual use. Let us illustrate them with the example of a database of English words organized so as to help in solving crossword puzzles. We write one word per card and index it according to which vowels it contains and which ones it does not contain. Across the top row of the card we punch 10 holes labeled $A, E, I, O, U,$ $\sim A, \sim E, \sim I, \sim O, \sim U$. When a word, say *ABACA*, exhibits a given vowel, such as A, we cut a notch above the hole for A; when it does not, such as E, we cut a notch above the hole for $\sim E$ (pronounced "not E"). The figure below shows the encoding of the words *BEAUTIFUL, EXETER, OMAHA, OMEGA*. For example, we search for words that contain at least one E, but no U, by sticking two needles through the pack of cards at the holes E and $\sim U$. *EXETER* and *OMEGA* will drop out. In principle it is easy to make this sample database more powerful by including additional attributes, such as "A occurs exactly once", "A occurs exactly twice", "A occurs as the first letter in the word", and so on. In practice, a few dozen attributes and thousands of cards will stretch this mechanical implementation of a multikey data structure to its limits of feasibility.

The reader might be interested in working out the logic of evaluating queries expressed as arbitrary Boolean expressions over these attributes, observing that *AND* works in parallel with multiple needles, whereas *OR* is processed sequentially using multiple passes.

Exotic as the physical realization may appear today, the logical structure of edge-notched cards is an amazingly modern example of multi-key data structures: it organizes the search space as a hypercube, i.e. a regular grid in which every key value, whether present or not among the actual data to be stored, has its predefined place. Moreover, the physical order of the data to be stored is entirely independent of the logical order imposed on the data

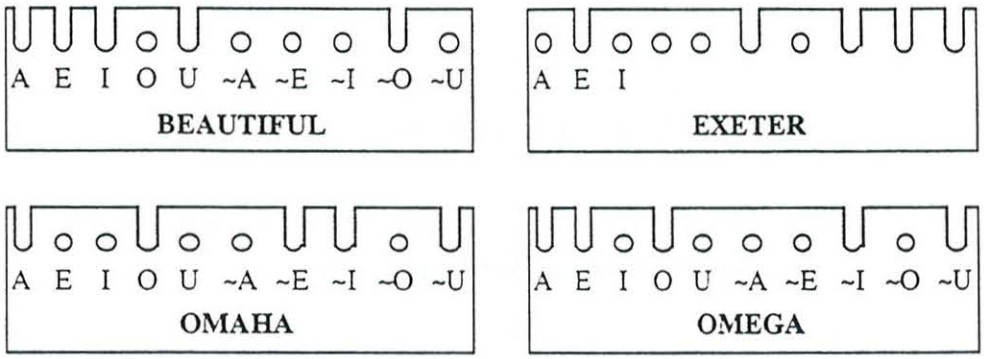

Figure 1: Edge-notches cards as a mechanical multi-key access structure

space, so the former can be chosen to match the physical characteristics of the hardware used (e.g. allocation of clusters of cards to boxes or, today, of records to disk blocks) without interfering with the search logic. But in order to fully appreciate these remarks we must describe the major evolutionary steps along the way from the first computerized data structures to modern multi-key data structures. We will observe that each era that focused on new application domains created new techniques to address concerns not adequately met by prior developments.

2.2 Early scientific computation: Static data sets

Numerical computation in science and engineering mostly leads to linear algebra and hence matrix computations. Matrices are static data sets: The values change, but the shape and size of a matrix rarely does - this is true even for most sparse matrices, such as band matrices, where the propagation of nonzero elements is bounded. Arrays were Goldstine and von Neumann's answer to the requirement of random access, as described in their venerable 1947 report "Planning and coding of problems for an electronic computing instrument". FORTRAN '54 supported arrays and sequential files, but no other data structures, with statements such as DIMENSION, READ TAPE, REWIND, and BACKSPACE.

Table look-up was also solved early through hashing. The software pioneers of the first decade did not look beyond address computation techniques (array indexing and hashing) because memories were so small that any structure that "wastes" space on pointers was considered a luxury. Memories containing a few K words restricted programmers to using only the very simplest of data structures, and the limited class of problems addressed let them get away with it. The discipline of data structures had yet to be created.

2.3 Commercial data: Batch processing of dynamic sets, single key access

Commercial data processing led to the most prolific phase in the development of data structures. The achievements of the early days were comprehensively presented in Knuth's pioneering books on "The Art of Computer Programming" (Knuth 1968, 1973). These applications brought an entirely different set of requirements for managing data typically organized according to a single 'primary key'. When updating an ordered master file with unordered transaction files, sorting and merging algorithms determine data access patterns. The emergence of disk drives extended the challenge of data structure design to secondary storage devices. Bridging the 'memory-speed gap' became the dominant practical problem. Central memory and disk both look like random access devices, but they differ in the order of magnitude of two key parameters:

	Memory	Disk	Ratio: Disk/Memory
Access time (seconds):	$10^{-7} \ldots 10^{-6}$	$10^{-2} \ldots 10^{-1}$	$10^{4} \ldots 10^{5}$
Size of transfer unit (bits):	$10 \ldots 10^{2}$	$10^{4} \ldots 10^{5}$	$10^{2} \ldots 10^{3}$

In recent decades technology has reduced both time parameters individually, but their ratio has remained a 'speed gap' of about 4 orders of magnitude. This fact makes the **number of disk accesses** the most relevant performance parameter of external data structures. Many data structures perform well in central memory, but disk forces us to be more selective; disks call for data structures that avoid pointer chains that indiscriminately cross disk block boundaries. The game of designing data structures suitable for disk has two main rules: the easy one is to use a small amount of central

memory effectively to describe the current allocation of data on disk in a way that facilitates rapid retrieval; the hard one is to ensure that the structure adapts gracefully to the ever-changing content of the file.

Index-sequential access methods (ISAM) order records according to a single key so that a small directory, preferably kept in central memory, ideally directs any point query to the correct data bucket where the corresponding record is stored, if it is present at all. But the task of maintaining this single-disk-access performance in a dynamic file, in the presence of insertions and deletions, is far from trivial. The first widely used idea, of splitting storage into a primary area and an overflow area, suffers from now well-known defects.

Balanced trees, one of the major achievements of data structure design, provide a brilliant solution to the problem of 'maintaining large ordered indexes' without degradation. They come in many variations (e.g. Adelson-Velski et al. 1962, Bayer et al. 1972) all based on the same idea: Frequent small rebalancing operations that work in logarithmic time eliminate the need for periodic reorganization of the entire file. Trees based on comparative search derive their strength from the ease of modifying list structures in central memory. They have been so successful that we tend to apply and generalize them beyond their natural limitations. In addition to concerns about the suitability of comparative search trees for multikey access, discussed in the next section, these limitations include (Nievergelt et al. 1981):

1. The number of disk accesses grows with the height of the tree. Depending on the size of the file and the fan-out from a node, or from a page containing many nodes, the tree may well have too many levels for instantaneous retrieval.

2. Concurrency. Every node in a tree is the sole entry point to the entire subtree rooted at that node, and thus a bottleneck for concurrent processes that pass through it, even if they access different physical storage units. Early papers (e.g. Bayer et al. 1977, Kung et al. 1980) showed that concurrent access to trees implemented as lists requires elaborate protocols to insure integrity of the data.

2.4 Transaction processing: Interactive multikey access to dynamic sets

Whereas single-key access may suffice for batch processing, transaction processing, as used in reservations or banking systems, calls for multikey access (by name, date, location, etc.). The simplest ideas were tried first. Inverted files try to salvage single-key structures by ordering data according to a 'primary key', and 'inverting' the resulting file with respect to all other keys, called 'secondary'. Whereas the primary directory is compact as in ISAM, the secondary directories are voluminous: Typically, each directory has an entry for every record. Just updating the directories makes insertion and deletion time-consuming.

Comparative search trees enhanced ISAM by eliminating the need for overflow chains, so it was natural to generalize them to multikey access and improve on inverted files. This is easy enough, as first shown by *k-d* trees (Bentley 1975). But the resulting multi-key structures are neither as elegant nor as efficient as in the single-key case. The main hindrance is that no total order can be imposed on multidimensional space without destroying proximity relationships. As a consequence, the simple rebalancing operations that work for single-key trees fail, and rebalancing algorithms must resort to more complicated and less efficient techniques, such as general dynamization (Willard 1978, Overmars 1981).

Variations and improvements on multidimensional comparative search trees continue to appear (e.g. Lomet et al. 1989, 1990). Their main virtue, acceptable worst case bounds, comes from the fact that they partition the actual data to be stored into (nearly) equal parts. The other side of this coin is that data is partitioned *regardless of where in space it is located.* Thus the resulting space partitions exhibit no regularity, in marked contrast to radix partitions that organize space into cells of predetermined size and location.

2.5 Knowledge representation: Associative recall in random nets

There is a class of applications where data is most naturally thought of as a graph, or network, with nodes corresponding to entities and arcs to relationships among these. Library catalogs in information retrieval, hypertexts with their many links, semantic nets in artificial intelligence are examples.

The characteristic access pattern is 'browsing': A probe into the net followed by a walk to adjacent nodes. Typically, a node is not accessed because of any inherent characteristic, but because it is associated with (linked to) a node currently being visited. The requirements posed by this type of problem triggered the development of list processing techniques and list processing languages.

These graphs look arbitrary, and the access patterns look like random walks - neither data nor access patterns exhibit any regular structure to be exploited. The general list structures designed for these applications have not evolved much since list processing was created, at least not when compared to the other data structures discussed. The resulting lack of sophisticated data structures for processing data collections linked as arbitrary graphs reminds us that efficient algorithms and data structures are always tailored to the presence of specific properties to be exploited, in particular to a regular structure of the data space. If the latter has no regular structure to be exploited, access degrades to exhaustive search.

Information retrieval is an example of an application where "random" structure cannot be avoided. Although a catalog entry in a library is a record with a regularly structured part, e.g. document = (author, title, publisher, year), search by content relies on index terms chosen from a large thesaurus of thousands of concepts, ranging from "Alchemy" to "Zen". It does not help to consider 1000 index terms, each with a range of perhaps only two values "relevant" and "irrelevant", as attributes of a multi-key data structures, which typically are designed to handle at most tens of access keys.

2.6 Spatial data management: Proximity access to objects embedded in space

In typical applications that rely on spatial data, such as computer-aided design or geographic information systems, many or all of the requirements listed so far are likely to appear: interactive transaction processing, random-looking networks of references among functionally related components, etc. In addition to such non-spatial requirements, spatial data imposes three key characteristics that sets spatial data management apart from the cases described above:

1. Data represents objects embedded in some d-dimensional Euclidean space $\mathbb{R}^d$.

2. These objects are mostly accessed through their location in space, in response to a proximity query such as intersection with some query region, or containment therein.

3. A typical spatial object has a significantly more complex structure than a 'record' in the other applications mentioned.

Although other applications share some of these characteristics to a small extent, in no other do they play a comparably important role. Let us highlight the contrast with the example of a collection of records, each with two attributes, 'social security number' (SSN) and 'year of birth'.

1. Although it may be convenient to consider such a record to be a point in a 2-d attribute space, this is not a Euclidean space; the distance between two such points, for example, or even the distance between two SSNs, is unlikely to be meaningful.

2. Partial match and orthogonal range queries are common in data processing applications, but more complex query regions are rare. In contrast, arbitrarily complex query regions are common in geometric computation (e.g. intersection of objects, or ray tracing).

3. Although a record in commercial data processing may contain a lot of data, for search purposes it is just a point. A typical spatial object, on the other hand, is a polyhedron of arbitrary complexity, and we face the additional problem of representing it using predefined primitives, such as points, edges, triangles, tetrahedra.

2.7 Concise summary of trends that shaped the development of data structures

Starting from the standard data processing task of the 50s and 60s: "merge the old master tape with an update file to produce a new master", we have mentioned a few of the many requirements that gradually accumulated a great variety of data handling problems. Here is a concise summary that compares yesterday's and today's requirements:

- Batch processing ⇒ interactive use:
  sequential access ⇒ random access
  static file ⇒ dynamic file
  delayed result ok ⇒ "instantaneous" response = 0.1 sec
- Simple queries ⇒ complex queries
  (e.g. access record with unique id ⇒ join in relational DB, proximity query in CAD)
  single key ⇒ multi-key access
  few query types ⇒ many different query types
  (e.g. point query ⇒ region query, consistency check, ..)
  single access ⇒ multi-access transactions
- Point objects ⇒ interrelated objects of arbitrary shape
  (e.g. [name, SSN, year] ⇒ assembly of mechanical parts)

Whereas the generic data structure 'array' was able to meet most needs of numerical computation for decades, it soon became evident that no single type of data structure could be found to meet the increasing variety of data handling problems that arose when computer use expanded to many other applications. Since the sixties, the search for specialized structures designed to handle efficiently a specific set of requirements has never ceased, and the resulting zoo of data structures is impressive, perhaps frightening to the non-specialist. The next section develops concepts needed to detect some order in the wilderness of data structures.

3 Basic concepts and characteristics of multi-dimensional and spatial data structures

Having presented a concise historical survey that introduced many ideas needed to understand data structures in general, we now narrow the conceptual framework to deal with multi-dimensional data, and spatial structures in particular. This section aims to be a user's guide to understand the zoo of data structures.

3.1 The profile of a data structure

A relatively small number of concepts suffice to characterize any data structure according to its main features, to highlight similarities and contrasts with other structures that might be considered as alternatives, and to guide a programmer in his choice of data structure. These key concepts surface when answering the following questions:

- What type of data is to be stored? This is answered by specifying the **domain** D of key values, and all operations and relations defined on D. Well-known frequent cases include :
  - D is an unordered set, such as author-defined index terms for document retrieval
  - single-key: D is totally ordered w.r.t. a relation "$<$" (e.g. integers, character strings)
  - multi-key: D is a Cartesian product $D_1 \times D_2 \times \ldots \times D_k$ of totally ordered domains D_i.

  When a meaningful distance is defined on D, we talk about "metric data structures". The most prominent example is Euclidean space $\mathbb{R}_1 \times \mathbb{R}_2 \times \ldots \times \mathbb{R}_k$, where $\mathbb{R}$ denotes real numbers, or perhaps integers.

- How many keys (search attributes) are involved? The **dimension of the space** has a great influence on the practical complexity of a multi-key data structure. Some approaches that work well in 2 dimensions, for example, do not generalize efficiently to more dimensions. One might think that "spatial data" obviously refers to *2-d* and *3-d* Euclidean space, but this is not necessarily so. Higher-dimensional spaces arise naturally when we describe objects to be stored in terms of parameters that characterize them. The term "multi-key access", including spatial data, commonly refers to the case where we have less than 10 keys (search attributes).

- Functionality (Abstract Data Type): What operations must be supported by the data structure? The most frequently used data structures are variations of the type **dynamic table or dictionary**. They support primarily the operations **Find, Insert, Delete**, along with a

host of others such as Predecessor, Successor, Min, Max, etc. Many algorithms that work on spatial data naturally use standard data structures such as stacks and queues, but there is nothing "spatial" about these.

- **How much data** is to be stored, **what storage media** are involved? The two major categories to be distinguished are internal data structures, designed for central memory, and external ones, designed for disk. Many more designs are suitable for internal data structures than for external ones.

- What type of **objects** are to be stored, how simple or complex is their description? Points are certainly the simplest case. Complex objects are often approximated or packaged in simple containers, such as a bounding box. Is the location of these objects fixed, or are they movable or subject to other transformations? In the second case it is important to separate the description of the object from its location.

- What types of **queries** occur, what do we know about their frequency and relative importance? This involves differences such as interactive or batch processing, clustered or scattered access, exact or approximate matches, and many more. Only rarely can we characterize the query population in terms of precise statistical parameters.

- How complex, how efficient are **access and update algorithms**? The efficiency of internal data structures is often well described by their asymptotic time complexity (e.g. $O(1)$, $O \log n$, $O(n)$, ...), whereas that of external data structures is more meaningfully measured in terms of the (small) number of disk accesses needed.

- What **implementation techniques** are appropriate (e.g. lists, address computation)? List processing is a prime candidate for dynamic data structures in central memory, but is often less efficient for external data structures.

3.2 The central issue: Organizing the embedding space versus organizing its contents

The single most important issue that distinguishes spatial data structures from more traditional structures can be summarized in the phrase "organizing the embedding space versus organizing its contents". Let us illustrate this somewhat abstract idea with examples.

A record with fields such as (name, address, social security number, ...) can always be considered to be a point in some appropriate Cartesian product space. But the role and importance of this **embedding space** for query processing, whether its structure is exploited or not, depends greatly on the application and the nature of the data under consideration. Whereas the distance between two character strings, say your address and mine, has no practical relevance, the distance between two points in Euclidean space often carries information that is useful for efficient query processing. In addition, regions in Euclidean space admit relations such as "in front of", "contains", "intersects" that have no counterpart in non-spatial data. For this reason, the embedding space plays a much more important role for spatial data than for any other kind of data.

Early data structures, developed for non-spatial data, could safely ignore the embedding space and concentrate on an efficient organization of the particular data stored at any one moment. This point of view naturally favored **comparative search** techniques. These organize the data depending on the relative value of these elements to each other, regardless of the absolute location in space of any individual value. Comparative search (e.g. binary search) leads to structures that are easily balanced. Thus, they answer **statistical queries** efficiently (e.g. median, percentiles), but **not general location queries** ("who is closest to a given query point", "where are there data clusters"). Balanced trees, with their logarithmic worst-case performance for single-key data, are the most successful examples of structures that organize a specific data set.

Given the success of comparative search for non-spatial data, in particular for single-key access, it is not surprising that the first approaches to spatial data were based on them. And that the crucial role of the embedding space, independently of the data to be stored, was recognized rather late. But when comparative search is extended to multi-dimensional spatial data, some shortcomings cannot be ignored. If we generalize the idea of a balanced

binary search tree to 2 dimensions, as in the following example, we generate a space partition that lacks any regularity. Such a partition does not make it easy to answer the question what cells of the partition lie within a given query region. Even the idea of dynamically balancing the tree, so as to guarantee logarithmic height and access time in the presence of insertions and deletions, does not generalize efficiently from 1 to 2 dimensions.

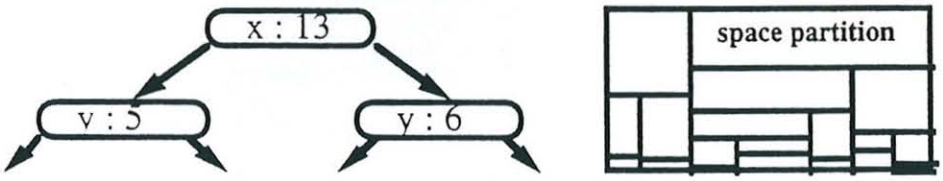

Figure 2: By balancing data, k-d trees generate irregular space partitions of great complexity

Data structures based on regular radix partitions of the space, on the other hand, organize the domain from which data values are drawn in a systematic manner. Because they support the metric defined on this space, a prerequisite for efficient query processing, we call them metric data structures. The essential structure of these space partitions is determined before the first element is ever inserted, just as inch marks on a measuring scale are independent of what is being measured. The actual data to be stored merely determines the granularity of these regular partitions. Like the "longitude-latitude" partition of the earth, they use fixed points of reference, independent of the current contents. Thus any point on earth has a unique permanent address, regardless of its relation to any cities that may or may not be drawn on a current map.

The well-known quad-tree (Finkel et al. 1974, Samet 1990 a, b) illustrates the advantages of a regular partition of the embedding space, in this example a unit square. A hierarchical partition of this square into quadrants and subquadrants, down to any desired level of granularity, provides a general-purpose scheme for organizing space, a skeleton to which any kind of spatial data can be attached for systematic access. The picture below shows a quarter circle digitized on a $16 \cdot 16$ grid, and its compact representation as a 4-level quadtree.

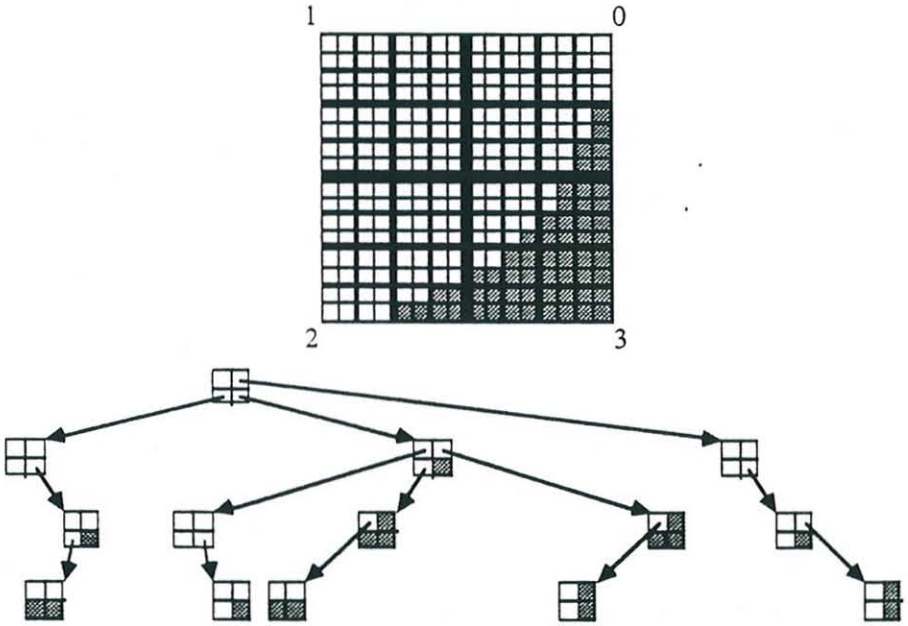

Figure 3: The quad tree is based on a radix 4 hierarchical space partition

Most queries about spatial data involve the absolute position of objects in space, not just their relative position among each other. A typical query in computer graphics, such as a visibility computation by means of ray tracing, asks for the first object intercepted by a given ray of light. Computing the answer involves absolute position (location of the ray and objects) **and** relative order (nearest along the ray). Regular space partitions reduce search effort by providing (direct) access to any cell of the partition. Given a point with coordinates (x, y), a simple formula determines the unique index of the unique cell (at any given level of granularity) that contains (x, y). Moreover, if storage is allocated contiguously as illustrated in Fig. 4, the address of the disk block of each cell can also be computed merely on the basis of the

coordinates (x, y). This allows for direct access to those objects that are near to any query point.

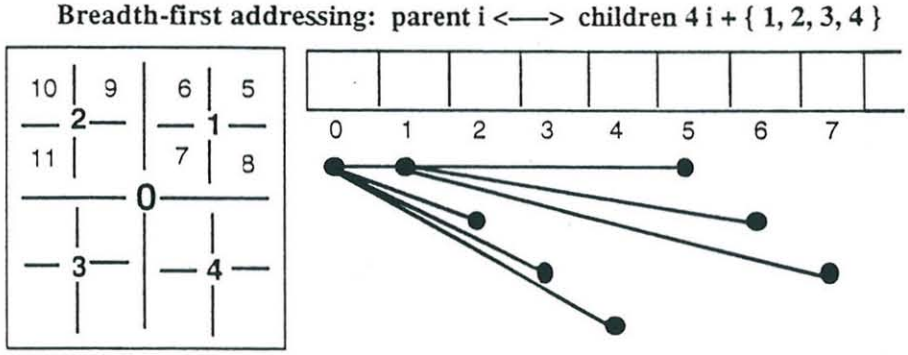

Figure 4: Breadth-first traversal allocates quad tree cells contiguously in order of increasing depth

4 The three step model of spatial object retrieval

Having surveyed the main concepts the reader needs to keep in mind when exploring the space of spatial data structures, we can now compress this information into a single model that captures the essence of how most spatial data structures process the vast majority of queries. Naturally, many details remain to be filled in to define precisely how the three processing steps outlined below are implemented for any given data structure. But the point to be made is that spatial query processing is best described in terms of three steps that can be analyzed independently.

Consider the embedding space shown in Fig. 5. It is grid-partitioned into rectangular cells and populated by objects drawn as circles or ovals. A triangular query region q calls for the retrieval of all objects that intersect q. In this typical example, query processing first transforms the query q into the set of query cells surrounded by the dotted line; second, retrieves

the two objects that intersect the query cells (without having to look at the horizontal oval); third, filters out the tall oval as a "false drop", and retains the circle as a "hit".

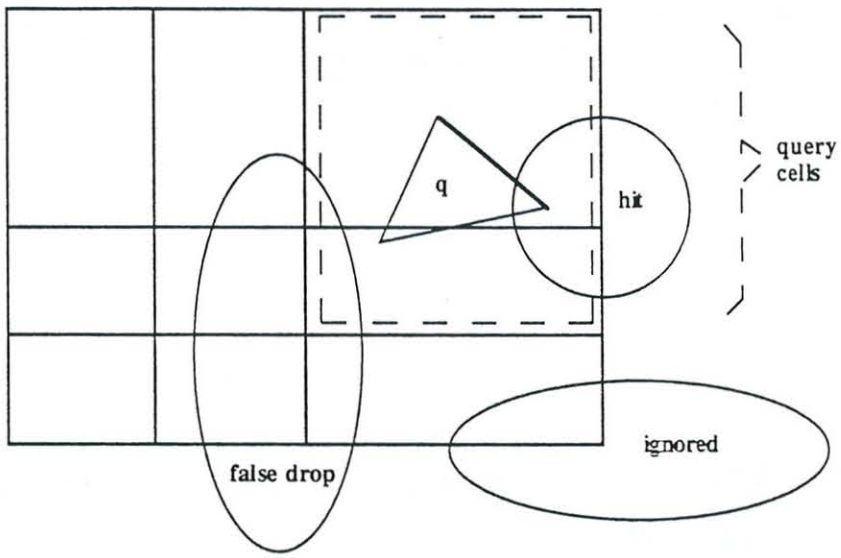

Figure 5: A region query selects objects that populate a grid-partitioned space

A more general description of these three processing steps follows:

4.1 Cell addressing: query $q \rightarrow$ set $\{(i, j, \ldots)\}$ of query cells

Obtain the coordinates of all cells, at any desired hierarchical level, of those cells that intersect q. This address computation step depends on characteristics of the query and of the grid, but **not** directly on the population of objects. The objects affect this step only to the extent that they determine the current space partition. Thus a simple space partition, i.e. a grid

of regular structure determined by a few parameters, is a pre-condition for fast cell addressing, and this is perhaps the data structure designer's major choice. He may have less control over the set of permissible queries, but fortunately query complexity is a lesser problem than partition complexity. Even if a complex query causes much computation, it need not cause any disk accesses, if the space partition has been properly designed so as to be completely described by a small amount of data that resides in central memory. And given that disk access is the efficiency bottleneck of spatial data structures, the time required by this first step is generally negligible.

4.2 Coarse filter: set of query cells determines candidate objects

All the objects that populate the query cells determined in step 1 are retrieved from disk, because they might respond to the query. This coarse filter is the bottleneck of spatial data access, and the core problem of data structure design. Many issues must be resolved, e.g: what is the precise definition of "an object O populates cell C?" The picture above suggests the plausible definition "O intersects C", and if so, this coarse filter retrieves the circle and the vertical oval, a hit and a false drop, whereas the horizontal oval is ignored. More sophisticated choices are possible that avoid associating an object with all of the many cells it might intersect, but each choice requires corresponding retrieval algorithms to ensure that no objects are missed. And the main issue of data structure design revolves around the association of data buckets (disk blocks) to cells, where the aim is to allocate objects that touch neighboring cells in as few buckets as possible. However this coarse filter is designed, the disk accesses it may cause are likely to require the lion's share of query processing time.

4.3 Fine filter: compare each object to the query

The objects selected by the coarse filter are mere candidates that need a final check to see whether they are hits, i.e. respond to the query, or false drops, i.e. passed the first but failed the second, crucial test: Does an object that appears to be close enough at first sight, really intersect the query? This intersection test is trivial or complex depending on the shape and complexity of query and object. But floating point operations are cheap compared to

disk accesses, so this fine filter is unlikely to be the performance bottleneck. In any case, this step is squarely in the realm of computational geometry. Its implementation depends on the internal representation of the objects, i.e., on the data structure chosen to represent an object for processing in central memory. A complex object will have to be broken into its constituent parts, such as vertices, edges, and faces in the case of a polyhedron. This has little to do with the design of the external data structure - the topic of our survey - which considers an object as a volume of space, to be treated as an undivided unit whenever possible.

A final comment: Everything discussed above also applies to the case where the "objects", drawn as ovals in the picture, are containers, chosen for their simple shape, that hold a more complex object. In this case the circle, which was labeled a "hit", is reduced to a mere "container hit", and the fine filter must process the object hidden inside, rather than the container.

As we discuss the design choices that characterize the many spatial data structures described in subsequent sections the reader is encouraged to keep the first two query processing steps in mind: how fast is cell addressing? how is the query cell population determined, and what disk accesses are caused by the coarse filter? Such questions serve as a guide for a first assessment of any spatial data structure. Often, they suffice to eliminate from further consideration apparently plausible ideas that fail on the grounds that the first or the second step cannot be implemented efficiently.

5 A sample of data structures and their space partitions

5.1 General consideration

Data structures for external storage support spatial queries to a set of geometric objects by realizing a fast but inaccurate filter: The data structure returns a set of external storage blocks that together contain the requested objects (hits) and others (false drops). Thereafter, a fine filter analyzes each object retrieved to either include or exclude it from the response. The purpose of a data structure is to associate each object with a disk block in such a way that the required operations are performed efficiently.

For exact match, insert and delete operations, non-spatial data structures

such as the B-tree or extendible hashing are sufficient, since a unique key can be computed from the geometric properties of each object. But whenever a query involves spatial proximity, as in range queries, a spatial data structure must take into account the shape and location of objects. Naturally, a query can be answered faster if the set of geometric objects that form the response to the query are spread over as few disk blocks as possible. This implies that for proximity queries, the objects stored in a block of an efficient data structure should be close in space. As a consequence, spatial data structures cover the data space (or a part of it) with cells and associate a storage block with each cell. For point objects, the cells partition (a part of) the space, and each point is associated with the cell in which it lies. Our illustrations and explanations of data structuring concepts always refer to 2-dimensional data, but generalization to higher dimensions is often straightforward.

Our sample of spatial data structures is biased towards those suitable for external storage. This bias favors simple structures and penalizes sophisticated structures that use complicated lists. Thus, we exclude trees designed to support worst-case efficient algorithms in computational geometry, such as segment trees and interval trees, including their variants for disk storage. Segment trees and interval trees, as well as hierarchies of such trees making them multidimensional, where e.g. each node of a segment tree references an interval tree, have been studied extensively in computational geometry (Preparata et al. 1985, Edelsbrunner 1982, Iyengar et al. 1988, Overmars 1983, Samet 1988, Samet 1990a, van Kreveld 1992). Based on the segment trees designed for central memory and a worst case scenario, external storage structures have been proposed (Blankenagel 1991). They turned out to be quite complicated, and there is no evidence yet as to whether these external storage segment trees will perform well on average in practical situations. Among the simple structures we discuss, we emphasize address computation techniques, because they are conceptually the simplest and the easiest to implement, and they often lead to the most efficient access structures for practical situations.

Two dominant factors guide the partition of the embedding space into cells, namely the data (the objects) and the space. At one end of the spectrum, the partition is defined without attention to the data; at the other, the data completely determines the partition; naturally, combinations of the two abound.

5.2 Space driven partitions: Multidimensional linear hashing, space filling curves

The most regular partitions of the data space are those that take into account only the amount of data to be stored (measured by the number of objects, or by the number of storage blocks needed), but disregard the objects themselves and their specific properties. The number of objects merely determines the number of cells of the partition, but not their location, size or shape. The latter are inferred from a generic partition pattern for any number of regions that is parameterized with just one parameter, namely the actual number of regions desired. As a typical example, let us look at the partitions (Fig. 6) induced by multidimensional variants of linear hashing (Enbody et al. 1988, Litwin 1980), such as multidimensional order preserving linear hashing with partial expansions (Kriegel et al. 1986) or dynamic z-hashing (Hutflesz et al. 1988a).

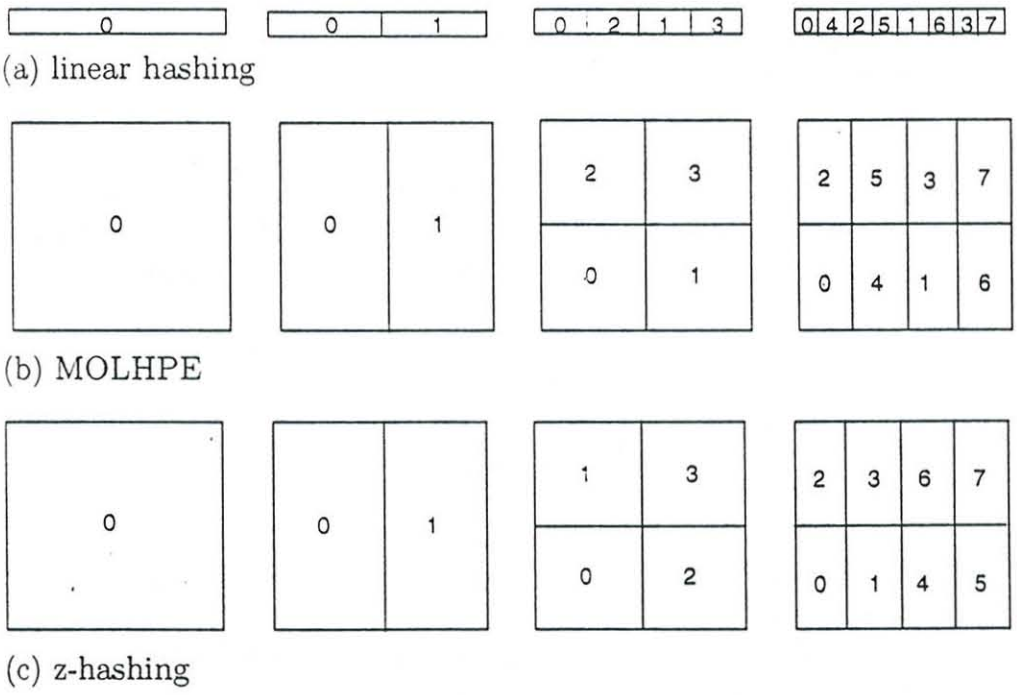

Figure 6: Space-driven partitions and their development

5.2.1 Linear hashing

When viewed as a spatial data structure, linear hashing (Fig. 6(a)) partitions the one-dimensional data space into intervals (one-dimensional regions) of at most two different sizes at any time, the smaller being half the larger. To the left of a separating position, all intervals are small; to the right, all intervals are large. This makes it very simple to find the interval in which a one-dimensional query point lies: Given the size of the smaller intervals and the separating position, a simple calculation returns the desired interval. With the use of an order-preserving addressing function, proximity in the 1-d data space is preserved in storage space. Dynamic modifications to the partition, induced by increasing or decreasing numbers of data points, are simple. An extra interval, for instance, is created by partitioning (splitting) the leftmost of the larger intervals into halves, distributing the data points associated so far with the split interval among the two new intervals, and adjusting the separating position. The separating position starts at the left space boundary and moves from left to right through the data space. On its move, it cuts the intervals encountered in half. After it has reached the right boundary of the data space, all intervals have been cut to the same size, thus the number of intervals has doubled (the doubling phase is complete), and the separating position is reset to the left boundary.

The simplicity of the partitioning pattern of linear hashing makes it extremely simple to keep track of the actual partition: the directory consists of only two values, one for the number (or size) of large (or small) intervals at the beginning of the current doubling phase, and one for the separating position. The cost for this simplicity due to the regularity of the partition is the lack of adaptivity of the partition to the actual data points. In general, it will be necessary to provide overflow blocks for intervals, since the number of data points in an interval can be larger than the block capacity. Whenever the data points are distributed evenly over the data space, the lack of adaptivity may be tolerable; otherwise, considerable inefficiency may result.

5.2.2 Multidimensional linear hashing

Multidimensional order preserving linear hashing versions are nothing but generalizations of linear hashing to higher dimensions. Therefore, they share the basic characteristics with linear hashing. They vary in the way the dimen-

sions are involved in the addressing mechanism. The most direct extension of linear hashing is multidimensional order preserving linear hashing with partial expansions (MOLHPE, Kriegel et al. 1986), where the doubling phases cycle through the dimensions. Starting with one block, the first doubling leads to two blocks separated in the first dimension, the second doubling leads to two more blocks, separated from the first two in the second dimension, and so on (see Fig. 6(b)).

5.2.3 Dynamic z-hashing

Since there is some freedom in choosing the addressing function, we can even extend the scope of our considerations and ask for mappings that preserve the geometric order of the data beyond blocks. Here, we request that regions of blocks whose addresses are close tend to be close in data space. This makes sense for proximity queries whenever it is faster to read a number of consecutive blocks than to read the same number of blocks, spread out arbitrarily on the storage medium; it has been used for writing in Wang et al. (1987). Since current disks typically have much higher seek plus latency times than transfer time, they qualify as good candidates. Similar in spirit, Dröge et al. (1993), Dröge (1995) investigate space partitioning schemes for variable size storage clusters instead of fixed size blocks. An addressing function that leads to a more global preservation of order is dynamic z-hashing (Hutflesz et al. 1988a, see Fig. 6(c)). The static version of this addressing mechanism (Manola et al. 1986, Orenstein et al. 1984, Orenstein 1989, 1990) is long known to cartographers as Morton encoding (Morton 1966); it is the same as the quad code (Samet 1990a) or the locational code (Abel et al. 1983, Tropf et al. 1981). One of its nice properties is the fact that addresses can be computed easily by interleaving the bits of the coordinates, cycling through the dimensions; therefore, the technique is also known as bit interleaving.

The only reason why closeness of blocks does not match closeness of cells precisely lies in the impossiblity of embedding a higher dimensional partition in a one-dimensional one while preserving distances. That is, when applied to one-dimensional linear hashing, dynamic z-hashing fully preserves global order.

5.2.4 Space-filling curves

Each of the above addressing mechanisms defines a traversal of the embedding space by visiting all cells in the order of their addresses, a so-called space-filling curve. A number of space-filling curves other than the ones above have been proposed, with the goal of maintaining proximity in space also in the one-dimensional embedding the curve defines. Since the data structure based on a space-filling curve must adapt the partition pattern dynamically, space-filling curves usually have recursive definitions. Fig. 7 shows two building blocks ((a) and (c)) and the three best-known space-filling curves based on them - bit interleaving (b), the Gray code (d) (Faloutsos 1985, 1988), and Hilbert's curve (e) (Faloutsos et al. 1989, Jagadish 1990b).

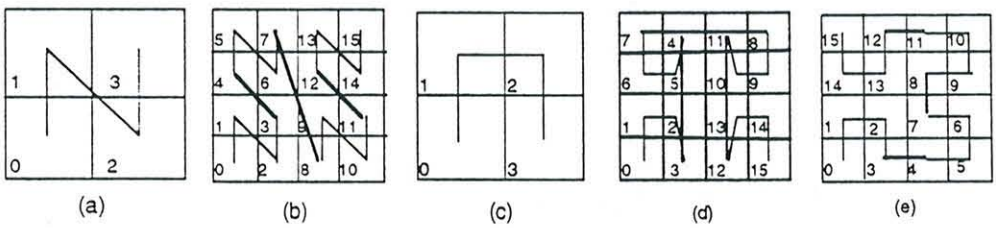

Figure 7: Traditional one-dimensional embeddings

Experiments comparing the efficiencies of data structures based on these curves (Abel et al. 1990, Jagadish 1990b, van Oosterom 1990) seem to indicate that for certain sets of geometric objects and sets of queries, bit interleaving and Hilbert's curve outperform the Gray code curve significantly, with Hilbert beating bit interleaving in many cases. On the other hand, an analysis of the expected behavior of space filling curves (Nievergelt et al. 1996), where all possible different queries are equally likely, indicates that all space filling curves are equally efficient, if disk seek operations are counted. This illustrates that there is a bias in the distribution of query ranges in the experiments: Queries are not chosen at random, but instead are taken to be in some sense typical of a class of applications. It is certainly useful to evaluate data structures with respect to particular query distributions; it is unfortunate that the distributions are not discussed explicitely. In contrast

to the average case, the worst case for hierarchical space filling curves clearly depends on the curve (Asano et al. 1995).

In spite of the interesting properties of dynamic z-hashing and other proximity preserving mappings of partitions in multidimensional space to one dimension, we feel that the importance of the corresponding data structures is limited to uniformly distributed data, due to the lack of adaptivity of the partition.

5.3 Data driven partitions: k-d-B-tree, hB-tree

The most adaptive partitions of all are those defined by the set of data points. Since the partition tends to be less regular, a mechanism to keep track of the partition is needed. A natural choice for such a mechanism is a hierarchy, and hence multidimensional generalizations of one-dimensional tree structures have been proposed for that purpose. Prime examples are the k-d-B-tree (Robinson 1981), a B-tree version (Bayer et al. 1972, Comer 1979) of the k-d-tree (Bentley 1975), and a modified version of it, the hB-tree (Lomet et al. 1989, 1990).

5.3.1 The k-d-B-tree

A k-d-B-tree partition is created from one region for the entire data space by recursive splits of regions (see Fig. 8). The splits follow the k-d-tree structure in that they cycle through the dimensions of the data space, but do so within each node of the tree. The leaves of the tree are all on the same level, just as in B-trees, and maintain the cells of the partition. Interior nodes maintain unions of cells to direct the search through the tree. Thus, the tree is leaf-oriented and serves as a directory. Whenever a data block overflows, due to an insert operation, its region is split so as to balance the number of data points in both subregions, and the change propagates towards the root. This may necessitate a split of a directory block region, not a simple operation in a k-d-B-tree. The reason is that in order to balance the load between both directory block subregions, a split position may be chosen that cuts through a region of some child (or even several children), thereby forcing the split to propagate downwards in the tree as well. Since the decision for the most balanced split position is made locally for a node, the forced downward split may become quite a costly operation, both in terms of runtime and of the

resulting storage space utilization. As a result, no lower bound on the storage space utilization can be given for k-d-B-trees, in contrast to the 50 guarantee for B-trees.

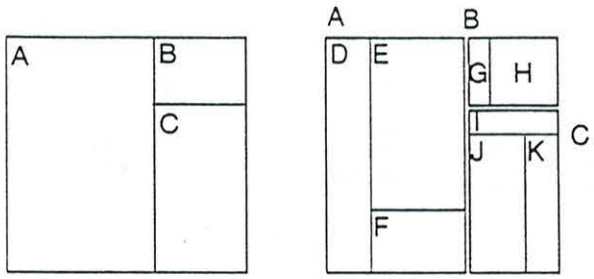

Figure 8: Data-driven partitions: The *k-d*-B-tree

5.3.2 The hB-tree

A closer look reveals that in degenerate cases, a balanced split may be impossible in the k-d-B-tree (see Fig. 9). To remedy that situation, a variant of the k-d-B-tree has been proposed, the hB-tree (Lomet et al. 1989, 1990). It has the interesting property that the regions that form the partition of the data space are not restricted to multidimensional rectangles. Instead, subregions can be removed from a region (see Fig. 9), leaving a "holey brick". With this freedom, balanced splits are possible in degenerate situations in which a single split line fails. The hB-tree keeps track of holey brick regions by allowing directory entries to refer to the same block: that is, a holey brick region is represented in the rooted DAG directory by the union of a set of rectangular regions. Of course, holey bricks may occur on each level of the rooted DAG; changes propagate upwards, just as in B-trees. As an example, Fig. 9c shows the rooted k-d-DAG local to a directory block to be split with regions shown in Fig. 9d, and Fig. 9b shows the part of the rooted DAG that propagates upwards when the block is split as shown in Fig. 9a.

Although data-driven partitions turn out to be quite complicated to maintain, they are able to cope with skew data reasonably well, while space-driven partitions fail here. For extremely skew data or whenever worst-case guarantees are more important than average behavior, data-driven partitions may be the method of choice. Due to the freedom in splitting regions, they certainly do have a great inherent flexibility that allows them to be tuned to

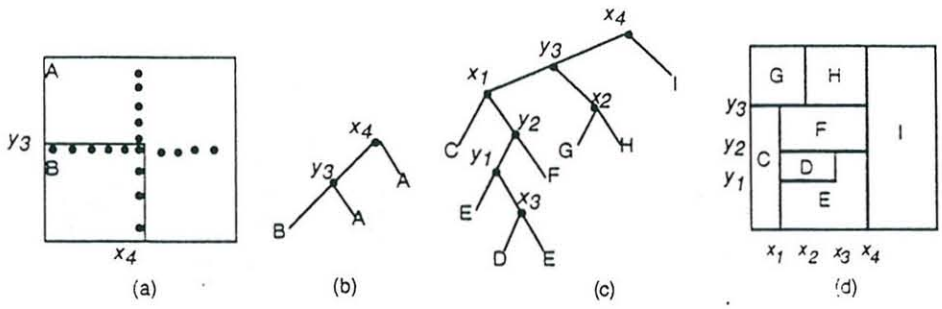

Figure 9: Data-driven partitions: The hB-tree

various situations easily. For instance, the local split decision (LSD-) tree is designed to make good use of a large available main memory, while resorting to external storage as necessary (Henrich et al. 1989a, b, Henrich 1990). Except for such situations, the adaptivity of data-driven partitions will rarely compensate for the burden of their complexity and fragility. More often, combinations of space-driven and data-driven partitions will be appropriate.

5.4 Combinations of space driven and data driven partitions: EXCELL, grid file, hierarchical grid files, BANG file

In their simplest form, these partitions follow the generic pattern of space-driven partitions, with the level of refinement determined by the data. A typical example of a one-dimensional data structure of this type is extendible hashing (Fagin et al. 1979), where the data space is partitioned by recursively halving exactly those subspaces that contain too many data points.

5.4.1 EXCELL and the grid file

A direct generalization of extendible hashing to higher dimension, EXCELL (Tamminen 1982), applies the one-dimensional strategy of extendible hashing to each dimension, again running cyclically through the dimensions (see Fig. 10a). Since an extra directory is available for each dimension, plus a

directory for the multidimensional product of the one-dimensional directories, and the number of blocks that can be addressed is therefore the product of the sizes of all one-dimensional directories, the sum of the sizes of the one-dimensional directories tends to be quite small in all realistic cases. This observation is used in the grid file (Nievergelt et al. 1981, 1984) to keep the directories for all single dimensions - the so-called scales - in main memory; as a result, no duplication of a directory is necessary due to a data block split, but instead a mere addition of a (multidimensional) directory entry will suffice (see Fig. 10b). Nevertheless, the grid file inherits from extendible hashing the superlinear growth of its directory (Regnier 1985, Tamminen 1985). A search operation in the grid file can always be carried out with just two external memory accesses, the first one to the directory, based on the information from the scales, and the second one to the data block referenced in the directory block. Similarly, a range query can be answered by first searching for a corner point of the query range, and then propagating to adjacent blocks as indicated by the directory. Since this entails a range query on the directory, one might organize the directory itself as a grid file.

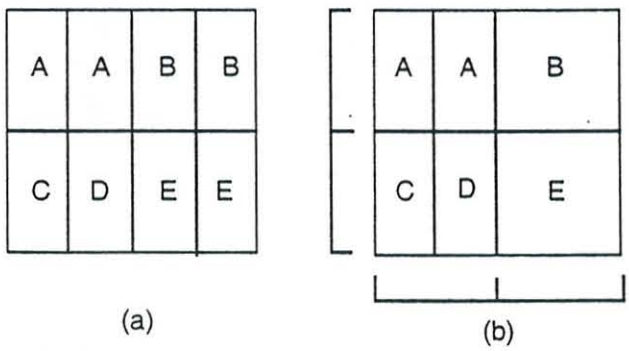

Figure 10: EXCELL and grid file partitions: Cells with data block addresses

5.4.2 Hierarchical grid files

This approach has been pursued in hierarchical grid files (Fig. 11) with two (Hinrichs 1985) or more directory levels (Krishnamurthy et al. 1985). The interior of each directory block is organized as a grid file directory; changes

in the tree structure, due to block split or merge operations, propagate along the search path in the tree. Data structures of this type are often called hash trees. A particularly efficient example of such a structure is the buddy tree (Seeger 1989, Seeger et al. 1990). It applies a specific merge strategy, and it distinguishes the standard block regions, using them for insertions, from the block regions used for all search operations: The latter are the bounding boxes of the objects stored in the corresponding subtree. While the non-hierarchical grid file inherits the property of a superlinearly growing directory from extendible hashing, hash trees grow linearly, just like trees in general. This does not imply that hash trees are always the better choice in practice: It is true for many data structures that better asymptotic efficiency may come at the cost of higher conceptual complexity and therefore lead to poorer performance in practical applications.

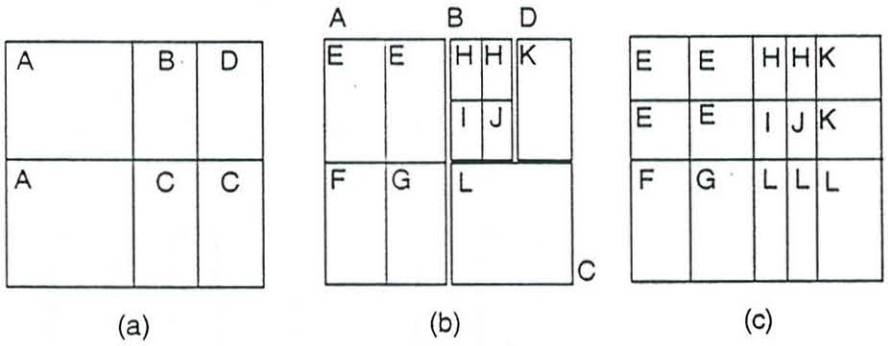

Figure 11: Two-level (a), (b) against one-level grid file partition (c)

5.4.3 The BANG file

The balanced and nested grid file (BANG file, Freeston 1987) is a particularly interesting attempt at balancing the load in a regular partition pattern. It operates on a regular grid in such a way as to guarantee linear growth of the directory. Unlike the grid file, the BANG file splits a cell such that the numbers of objects in the two resulting subspaces differ as little as possible. This is done under the restriction that the smaller subspace is created by recursively cutting the subspace to be divided along some dimension into two halves. In addition, the cuts run cyclically through all dimensions (see

Fig. 12a. where point numbers indicate the insertion order). As a consequence, the BANG file tends to use fewer cells than the grid file (Fig. 12c), but these cells have a more complex shape. Nevertheless, it is not a problem to maintain these cell: we simply maintain rectangular cells and associate a point with the smallest of all cells containing it (Fig. 12b). These cells can even be stored in a highly compressed form: Since they are created with a very strict mechanism, each cell can be stored as a pair of indices, where the first one indicates the number of recursive cuts and the second one is the relative number in some numbering scheme, for instance that of MOLHPE (Fig. 12b). Not surprisingly, the BANG file directory is organized as a tree, following the BANG file strategy recursively. This results in the necessity to propagate splits not only upwards in the hierarchy, but also downwards - the forced split phenomenon shared by quite a few hierarchical spatial access structures.

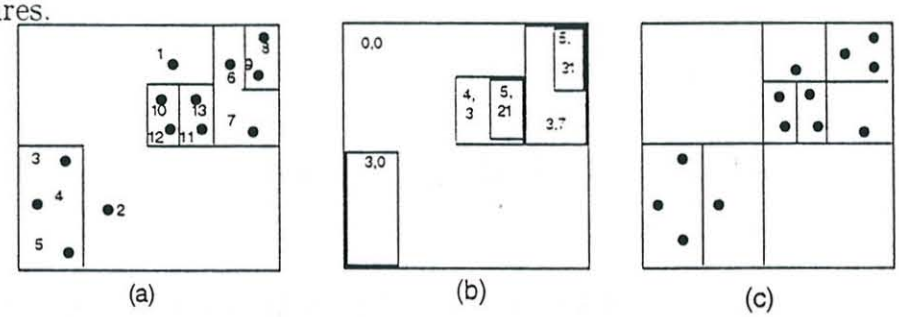

Figure 12: A BANG file partition (a), its maintenance (b), and a grid file partition (c) for capacity 3

There is a large number of data structures whose partition is driven by a combination of space and data considerations, with quad tree based structures (Finkel et al. 1974, Samet 1990a, b) being the most prominent ones among them. Others are variants of grid file or hash tree structures (Kriegel et al. 1988, Otoo 1986, 1990, Ouksel 1985, Ozkarahan et al. 1985) or adaptive hashing schemes (Kriegel et al. 1987, 1989b). Some of them aim in particular at high storage space utilization (Hutflesz et al. 1988b, c, d), apart from the efficiency of range queries.

5.5 Redundant data storage

So far we have presented data structures that store every data element (point) exactly once. This natural approach is universally followed in practice, because data redundancy complicates updating and therefore is used only to enhance reliability (e.g. back-up procedures), but is not part of the access structure.

From a theoretical point of view, however, one can ask whether replicating some part of the data might speed up retrieval. This turns out to be true in particular for static files. Chazelle (1990) proves the following lower bound for a pointer machine that executes static 2-d range searches: A query time of $O(t+\log^c n)$, where t is the number of points reported and c is some constant, can only be achieved at the expense of $\Omega(n \log n / \log \log n)$ storage.

Data structures that use data redundancy to improve access time for range searching include the P-range tree (Subramanian 1995), a combination of the priority search tree and a 2-d range tree.

6 Spatial data structures for extended objects

So far we have considered the simplest of spatial objects only, points, for the good reason that any spatial data structure must be able to handle point data efficiently. Most applications, however, deal with complex spatial objects. And although complex objects are composed of simpler building blocks, we encounter a multitude of the latter: line segments, poly-lines, triangles, aligned rectangles, simple polygons, circles and ovals, and the multi-dimensional generalizations of all these. The way a spatial data structure supports extended (non-point) objects of various kinds determines whether it is generally applicable. For extended objects, each of which may intersect a number of cells, the association of an object with a cell is not as immediate as it is for points. In this case, cells usually overlap, and an object is associated most often either with a cell that contains it, or with all cells it intersects. But there are also other possibilities. Therefore, while we distinguish data structures for points merely according to the type of cells they define, we characterize data structures for extended objects also according to the way they associate objects with regions.

We consider the case where objects to be stored are relatively simple, in the sense that they have a concise description, and computations are easy and efficient. This restriction is realistic because complex objects are often approximated or bounded by a simple container, such as a bounding box, the smallest aligned (axis-parallel) multidimensional rectangle that contains the object. Such a container serves as a conservative filter for spatial proximity queries. In a range query, for instance, an object intersects the query range only if its bounding box intersects the query range; similarly, a query point can be contained in an object only if it is contained in its bounding box. Most data structures that support extended objects limit themselves to aligned rectangles (bounding boxes); exceptions include (Bruzzone et al. 1993, Günther 1988, 1989, 1992a, Günther et al. 1989, 1991, Jagadish 1990a, van Oosterom et al. 1990). Even in this restricted case, it is by no means clear how to associate a rectangle with a region in space, because a rectangle may intersect more than one of these regions. There are essentially three different extremal solutions to this problem, and a fourth one as a combination of two extremes.

6.1 Parameter space transformations

Since points can be maintained in any of the ways described in the previous section, there is the obvious possibility to store simple objects as points in some parameter space. Simple mappings have been proposed that transform a d-dimensional rectangle into a 2-d-dimensional point (Henrich 1990, Hinrichs 1985, Seeger 1989). The corner transformation simply takes the 2-d rectangle boundary coordinates and interprets them as the coordinates of a point in 2-d space (see Fig. 13 a for d = 1). The center transformation separates parameters for the position of the rectangle in space from parameters for the size of the rectangle: the former are the d coordinates of the rectangle center, and the latter are the extensions of the rectangles in the d dimensions - divided by two, resulting in a more evenly populated data space (see Fig. 13 b). But then, fairly small query ranges (that seem to be common in practice) may map to queries with fairly large query regions (see Fig. 13 c and d), and the distribution of points, the data space partition and the shape of the query region seem not to work well together. For the special case in which range queries are the only type of proximity queries, good transformations have actually been found (Pagel et al. 1993a). These transformations use

parameters such as volume and aspect ratio, and they turn out to cluster rectangles in a way that is quite appropriate for many point data structures, especially in combination with highly adaptive space partitions such as the LSD-tree (Henrich et al. 1989b, Henrich 1990). Nevertheless, we feel that a transformation cannot be general enough to preserve the geometry of the situation entirely, and as a consequence, will not be able to support all kinds of locational queries well. In the remainder of this section, we will therefore look more closely at ways to store rectangles in the given space.

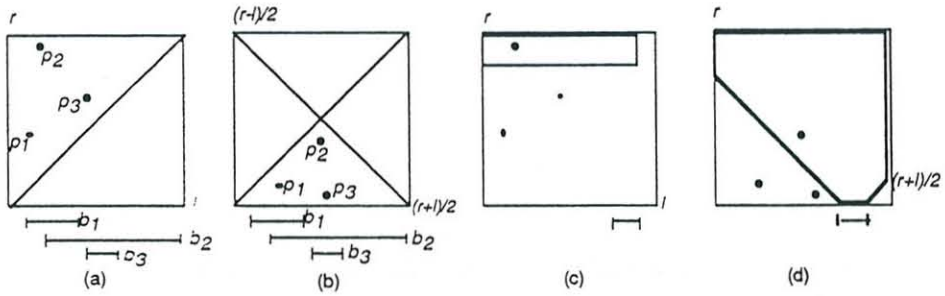

Figure 13: Corner transformation (a) and center transformation (b) for intervals and for query regions (c), (d)

6.2 Clipping

The problem in associating a rectangle with a region in a partition is the fact that a rectangle can intersect more than one region. For the technique of answering a range query by returning all those data blocks whose regions intersect the query range, there is no other choice but to associate each rectangle with all regions in the partition that it intersects. This method is called clipping. It can lead to reasonable performance in cases where the geometric object behind the bounding box can be cut at region boundaries, such as those cartographic applications in which objects are polygons with lots of corners (Schek et al. 1986, Waterfeld 1991). In these cases, the clipping technique has the advantage over many others to be conceptually simple and

to preserve the geometry of the situation for any proximity query. Clipping turns out not to lead to good performance for rectangles (Six et al. 1988), though, and it can become very bad in the worst case, with a linear proportion of all rectangles even on the best possible cut line (d'Amore et al. 1993a, b, Nguyen et al. 1993, d'Amore et al. 1995).

6.3 Regions with unbounded overlap: R-tree

6.3.1 Storing a reference point

In another straightforward way of using point data structures for storing objects, a reference point is chosen for each object - typically its center (of gravity) -, and the object is associated with the block in whose region its reference point lies. This works only if we keep track of the extensions of the objects beyond the region boundaries. In a range query, the cell to be considered for a block is not the cell defined by the partition of the data space, but instead the bounding box of all objects actually associated with the block. The latter may in general be larger than the former, and not all data structures will easily accomodate that extended region information. Hierarchical structures such as k-d-trees (Ooi 1987, Ooi et al. 1989) or the BANG file (Freeston 1989b) as well as some others (Seeger et al. 1988, Seeger 1989) can be used for that purpose. Even though this approach works well whenever only small objects are to be stored, it is inefficient in general, because no attempt is made to avoid the overlap of search regions, and therefore geometric selectivity is lost easily.

6.3.2 The R-tree family

With the explicit goal of high geometric selectivity, the R-tree (Guttman 1984) has been designed to maintain block cells that overlap just as much as necessary, so as to make each rectangle fall entirely within a cell. Its structure resembles the B+-tree, including restructuring operations that propagate from the leaf level towards the root. Each data cell is the bounding box of the rectangles associated with the corresponding block. Each directory block maintains a rectangular subspace of the data space; its block cell is the bounding box of the subspaces of its children. As a consequence, on each level of the tree, each stored rectangle lies entirely in at least one block cell;

since block cells will overlap in general, it may actually lie in more than one cell (in Fig. 14 , rectangles E and F lie in cells A and B, referenced by the root block). This fact may distort geometric selectivity: In an exact match query, we cannot restrict the search for E to either A or B, but instead we need to follow both paths (in the worst case, with no hint as to which one is more likely). Since it is essential for the R-tree to avoid excessive overlap of cells, many strategies of splitting an overflowing block into two exist, ranging from the initial suggestion (Guttman 1984) to less or more sophisticated heuristics (Beckmann et al. 1990, Greene 1989) and even to optimal splits for a number of criteria (Becker et al. 1992, Six et al. 1992). With the appropriate splitting strategy and extra restructuring operations (Beckmann et al. 1990), the R-tree seems to be one of the most efficient access structures for rectangles to date.

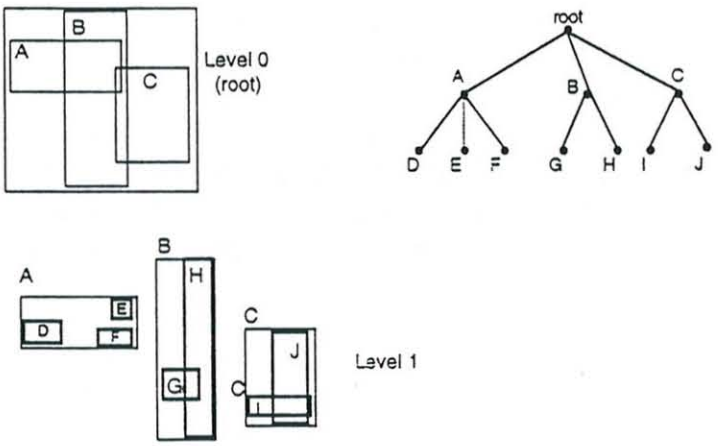

Figure 14: Cells of an R-tree

For specific purposes, a number of variants of the R-tree have been proposed. The R+-tree avoids overlapping directory cells by clipping rectangles as necessary (Faloutsos et al. 1987, Sellis et al. 1987). It suffers, however, somewhat from the inefficiency caused by forced splits propagating downwards, similar to k-d-B-trees. For a static situation, in which almost no

insertions or deletions take place, and therefore queries dominate the picture, the R-tree can be packed densely so as to increase effciency in time and space (Roussopoulos et al. 1985). Ohsawa et al. (1990) combine the R-tree and quad tree cells.

6.4 Cells with bounded overlap: multilayer structures, R-file, guard file

Geometric selectivity increases with decreasing cell overlap. This has been the starting point for a number of attempts to maintain rectangles with cells whose overlap is under tight control

6.4.1 The multilayer technique

The basic idea is to cover the data space with more than one partition. In the most direct way to implement this idea, each partition - called layer - is maintained in an extra data structure (Six et al. 1988). Care has to be taken to ensure that the partitions of the different layers are actually different (Fig. 15): otherwise, the number of partitions increases more than necessary, and in general efficiency deteriorates. A fairly large number of split strategies that guarantees the partitions to be quite different have been developed. It can be guaranteed that for storing a set of small d-dimensional rectangles, $d + 1$ layers will always suffice - the technical term small there has a precise meaning (Six et al. 1988). Large rectangles must be clipped, whenever there are few of them, such as in most cartographic applications, clipping will not be harmful.

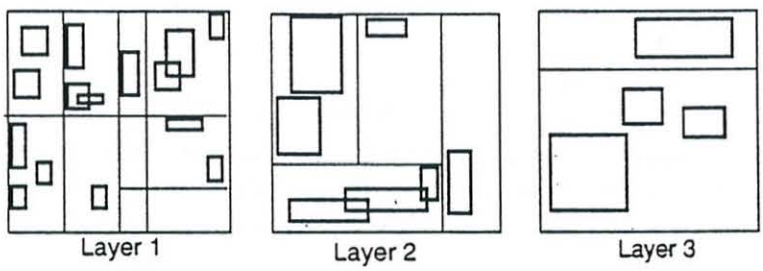

Figure 15: Cells of a three-layer grid file

In a multilayer data structure that uses a point data structure with a modified split strategy for each layer, the layers are totally ordered, from lowest to highest. A rectangle is associated with the lowest layer that has a cell of the partition containing it entirely; in that layer, the rectangle is associated with that cell, just like a point in the underlying point data structure. If there is no such layer, either a new layer is created, or the rectangle is clipped at the boundaries of the highest layer.

Note that the multilayer technique is generic in the sense that it allows any one of a number of point data structures to be used for the layers. Experiments with a multilayer grid file, for instance, show that the loss of efficiency as compared with a standard (single layer) grid file for points is tolerable, with a certain, but small, query overhead due to the fact that a directory for each layer needs to be inspected. Experiments with multilayer dynamic z-hashing (Hutflesz et al. 1992) show that the attempt to preserve global order makes range queries extremely fast, far better than any other data structure, for data and queries from cartography. In addition, the multilayer technique realizes the advantages of recursive linear hashing (Ramamohanarao et al. 1984) over linear hashing without the overhead, because there are several (recursive) layers anyway.

6.4.2 The R-file

Since the search parameters are the same for each layer of a multilayer structure (e.g. in a range query), it might be desirable to somehow integrate all the directories into just one directory. This is exactly what the R-file (Hutflesz et al. 1990) does: It maintains a set of overlapping cells with one directory. Since the grid file philosophy of data-space oriented partitioning and the BANG file technique for keeping the directory small have proven to be useful, both are applied in the R-file. Overlapping cells are defined by considering each cell to be a rectangle, exactly as in the BANG file, but objects are associated with cells in a different way. In the R-file, a rectangle is associated with the smallest cell that contains it entirely (Fig. 16). Superficially, this sounds to the BANG file technique, but since it is applied to rectangles instead of points, it creates overlapping rectangular cells instead of a partition into orthogonal polygons.

When too many rectangles are associated with one cell: if too many rectangles intersect the split line of a cell in which they lie, they cannot be

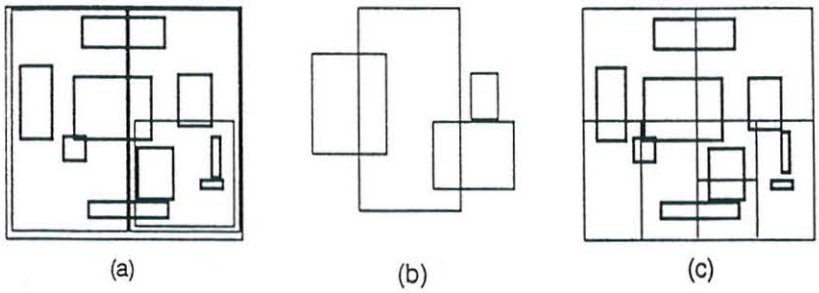

Figure 16: R-file cells for insertion (a) and for searching (b), and grid file cells with clipping (c)

distributed to other cells. In that case, a lower dimensional R-file stores these rectangles: the one dimension in which all of these rectangles intersect the split line can be disregarded. Fortunately, no extra data structure is needed for the (d-1)-dimensional R-file; instead, a simple skip of the corresponding dimension in the cyclic turn through all dimensions will do. Experiments have shown the R-file to be a very efficient data structure for cartographic data. It suffers, however, from the disadvantage of being somewhat complicated to implement, with extra algorithmic difficulties (but not inefficiencies) such as a forced split downwards. Nevertheless, it is a good basis for a data structure that supports queries with a requested degree of detail, all within a query range (Becker et al. 1991).

6.4.3 The guard file

The difficulties in maintaining objects, as opposed to points, comes from the fact that an object may intersect more than one cell of a space partition, that is, it may intersect split lines in a dynamic data structure. In the R-file, for instance, rectangles intersecting a split line are sometimes handled separately. The guard file deliberately places guard points (not lines) in strategic positions, and then distinguishes objects according to the guards they overlap. Nievergelt et al. (1993) study guard placements at the corners of the cells of some regular space partition. Fig. 17 shows a quad tree partition, but other partitions such as triangular or hexagonal ones work just as well.

By imposing a hierarchy on the guards, range queries become efficient: The level of a guard is the level of the largest cell of which the guard is a corner

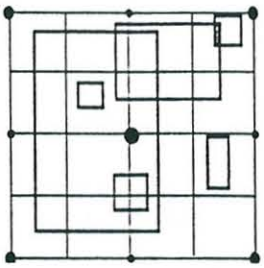

Figure 17: A quad tree partition with guards at corner points

point, with the root of the tree having the highest level. Any sufficiently large object must overlap a guard. An object containing a guard is stored with the highest level guard that it contains. An object that contains no guard must be small; it can be clipped and stored with all cells (of the partition, that is, at the leaf level of the quad tree) that it intersects. As a consequence, a range query would be carried out by inspecting the objects associated with all guards in the query range and all cells intersecting the query range. For long and skinny objects, clipping would make this data structure quite an inefficient one, but for fat objects (those with aspect ratio not too far from one) such as those often encountered in cartography, clipping would not be a problem.

The guard file, however, includes an additional step to trade storage space for query time: An unguarded object is associated with exactly one cell of the partition, namely the one into which its center (of gravity) falls. Hence, each object is stored only a small, constant number of times, the number depending only on the type of regular partition being used. In a range query, it is no longer sufficient to inspect the cells that intersect the query range: Adjacent cells also need to be inspected. The halo by which a query range must be extended depends entirely on how fat the stored objects are: the fatter the objects, the smaller the halo. For various types of regular partitions and various bounds on the fatness (in a precise, technical sense for arbitrary, convex geometric objects), it has been shown that the guard file is conceptually simple, easy to implement, and efficient at the same time.

6.5 Conclusions drawn from the structures surveyed

The set of spatial data structures and paradigms has grown to a respectable size. A multitude of concepts serve as building blocks in the design of new spatial data structures for specific purposes. A number of convincing suggestions indicate that, for instance, these building blocks can be used in the design of data structures for more complex settings. Becker (1992), Brinkhoff et al. (1993), Günther (1992b), Kriegel et al. (1992a), and Shaffer et al. (1990a), among others, show how various kinds of set operations, such as spatial join, can be performed efficiently by using a spatial index. For objects and operations going beyond geometry, Ohler (1992) shows how to design access structures that involve non-geometric attributes in a query. In a closer look to the needs of cartography, a data structure of weighted geometric objects is proposed that efficiently supports queries with a varying degree of detail, as specified by the user (Becker et al. 1991). For the special case in which the objects to be stored are the polygons of a planar partition, and access to entire polygons (in containment queries with points, for instance) as well as access to the polygon boundaries (for drawing a map) is needed, there are suggestions that consider both external accesses and main memory computations as important cost factors (Kriegel et al. 1992b, Schiwietz et al. 1993); other suggestions aim at making queries efficient without storing polygon boundaries twice (Becker 1993). In addition, hierarchical spatial access structures can be designed to support queries into the past (Becker et al. 1993, Charlton et al. 1990, Xu et al. 1990). It can be seen from this list that even though a fair number of concepts of data access are readily available, it is mosten often a considerable effort to bring any two of them together without losing key features, especially efficiency.

All the arguments above concerning the efficiency of data structures are of an intuitive nature. An average case analysis is rarely given (Ang et al. 1992, Devroye 1986, Flajolet et al. 1986, Lindenbaum et al. 1995, Nelson et al. 1987, Regnier 1985, Rottke et al. 1987 are notable exceptions), because a probability model for sets of geometric objects and queries is very hard to get (Ambartzumjan et al. 1993, Harding et al. 1974, Matheron 1975, Santalo 1976, Stoyan et al. 1987). Even experiments (Kriegel et al. 1989a, Shaffer et al. 1990b, Smith et al. 1990) that clarify some of the efficiency aspects tend to reveal not too much about the contribution of the building blocks of data structures to their overall efficiency. Some steps towards a clarification

of what is the desired average case efficiency and how to achieve it, with consequences for the data structure design, have been taken (Henrich et al. 1991, Pagel et al. 1993b, Six et al. 1992, Ang et al. 1992, Pagel 1995), but many more are needed to shed enough light on the inner workings of spatial data structures. In the meantime, the designer and programmer of spatial data structures should be aware of the existing building blocks and use them with expertise and intuition, bringing them together wherever possible.

7 Summary: Points to consider when choosing a spatial data structure

Let us conclude this bird's eye survey of the domain of spatial data structures with a few concisely stated points of view and recommendations.

Spatial data differs from all other types of data in important respects. Objects are embedded in a Euclidean space with its rich geometric structure, and most queries involve proximity rather than intrinsic properties of the objects to be retrieved. Thus data structures developed for conventional data base systems are unlikely to be efficient.

The most important feature of a spatial data structure is a systematic, regular organization of the embedding space. This serves as a skeleton that lets one address regions of space in terms of invariant quantities, rather than merely in relation to transient objects that happen to be stored. In particular, radix partitions of the space are more widely useful than data-driven space partitions based on comparative search.

The vast literature on spatial data describes a multitude of structures that differ only in technical details. Theory has not progressed to the stage where it can rank them in terms of efficiency. Performance comparisons representative of a wide range of applications and data are difficult to design, and reported results to this effect are often biased. Thus we are left to choose on the basis of common sense and conceptual simplicity.

Whereas the choice of a spatial data structure does not require a lot of detailed technical know-how, programming the computational geometry routines that implement the fine filter of Section 4.3 is a different matter. Writing robust and efficient procedures that intersect arbitrary polyhedra (say a query region and an object) is a specialist's task that requires a different set

of skills.

In conclusion, this survey is neither a "how-to" recipe book that describes the multitude of data structures, nor a sales pitch for any one in particular. It intends to be a thought-provoking display of the issues to ponder and to assess. Our message places responsibility for a competent choice where it belongs, with the programmer.

References

Abel, D.J., J.L. Smith (1983): A data structure and algorithm based on a linear key for a rectangle retrieval problem; Computer Vision, Graphics, and Image Processing, Vol. 24, 1-13

Abel, D.J., D.M. Mark (1990): A comparative analysis of some two-dimensional orderings; International Journal of Geographical Information Systems, Vol. 4, No. 1, 21-31

Adelson-Velskii, G.M., Y.M. Landis (1962): An algorithm for the organization of information, Doklady Akademia Nauk SSSR, Vol. 146, 263-266: English translation in: Soviet Math. 3, 1259-1263

Ambartzumjan, R.V., J. Mecke, D. Stoyan (1993): Geometrische Wahrscheinlichkeiten und Stochastische Geometrie, Akademie Verlag, Berlin

Ang, C.-H., H. Samet (1992): Average storage utilization of a bucket method: Technical Report, University of Maryland

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

Asano, T., D. Ranjan, T. Roos, P. Widmayer, E. Welzl (1995): Space filling curves and their use in the design of geometric data structures, Proc. Second Intern. Symp. of Latin American Theoretical Informatics LATIN '95, Valparaiso, Lecture Notes in Computer Science, Vol. 911, Springer-Verlag, 36-48

Bayer, R., C. McCreight (1972): Organization and maintenance of large ordered indexes; Acta Informatica, Vol. 1, No. 3, 173-189

Bayer, R., Schkolnick, M. (1977): Concurrency of operations on B-trees, Acta Informatica Vol. 1, 1-21

Becker, B. (1993): Methoden und Strukturen zur effizienten Verwaltung geometrischer Objekte in Geo-Informationssystemen, Dissertation, Mathematische Fakultät, Albert-Ludwigs-Universität, Freiburg, Germany

Becker, B., H.-W. Six, P. Widmayer (1991): Spatial priority search: An access technique for scaleless maps, Proc. ACM SIGMOD International Con-

ference on the Management of Data, Denver, 128-137
Becker, B., P. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, P. Widmayer (1992): Enclosing many boxes by an optimal pair of boxes; Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science STACS, Cachan, Lecture Notes in Computer Science, Vol. 577, 475-486
Becker, B., S. Gschwind, T. Ohler, B. Seeger, P. Widmayer (1993): On optimal multiversion access structures; 3rd International Symposium on Advances in Spatial Databases, Singapore, Lecture Notes in Computer Science, Vol. 692. Springer-Verlag, 123-141
Becker, L. (1992): A new algorithm and a cost model for join processing with grid files: Dissertation, Fachbereich Elektrotechnik und Informatik, Universität-Gesamthochschule Siegen, Germany
Beckmann, N., H.-P. Kriegel, R. Schneider, B. Seeger (1990): The R*-tree: An efficient and robust access method for points and rectangles; Proc. ACM SIGMOD International Conference on the Management of Data, Atlantic City, New Jersey, 322-331
Bentley, J.L. (1975): Multidimensional binary search used for associative searching: Communications of the ACM, Vol. 18, No. 9, 509-517
Blankenagel, G. (1991): Intervall-Indexstrukturen und externe Algorithmen für Nicht-Standard-Datenbanksysteme; Dissertation, FernUniversität Hagen
Brinkhoff, T., H.-P. Kriegel, B. Seeger (1993): Efficient processing of spatial joins using R-trees: Proc. ACM SIGMOD International Conference on the Management of Data, Washington D.C., 237-246
Brinkhoff, T. (1994): Der Spatial Join in Geo-Datenbanksystemen. Ph.D.thesis, Ludwig-Maximilians-Universität München.
Brinkhoff, T. und H.-P. Kriegel (1994): The impact of global clustering on spatial database systems. In Proc. 20th Int. Conf. on Very Large Data Bases, 168-179.
Brinkhoff, T., H.-P. Kriegel, and R. Schneider (1993): Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In Proc. 9th IEEE Int. Conf. on Data Eng., 40-49.
Brinkhoff, T., H.-P. Kriegel, R. Schneider, and B. Seeger (1994): Multi-step processing of spatial joins. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 197-208.
Bruzzone, E., L. De Floriani, M. Pellegrini (1993): A hierarchical spatial index for cell complexes; 3rd International Symposium on Advances in Spa-

tial Databases. Singapore, Lecture Notes in Computer Science. Vol. 692. Springer-Verlag, 105-122

Charlton. M.E., S. Openshaw, C. Wymer (1990): Some experiments with an adaptive data structure in the analysis of space-time data; Proc. 4th International Symposium on Spatial Data Handling, Zurich, 1030-1039

Chazelle. B. (1990): Lower bounds for orthogonal range searching: I. The reporting case; J. ACM Vol. 37, No. 2, 200-212

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

Comer. D. (1979): The ubiquitous B-tree; ACM Computing Surveys, Vol. 11, No. 2, 121-138

Crain, I.K. (1990): Extremely large spatial information systems: a quantitative perspective; Proc. 4th International Symposium on Spatial Data Handling. Zurich. 632-641

d'Amore. F., P.G. Franciosa (1993a): Separating sets of hyperrectangles. International Journal of Computational Geometry and Applications, Vol. 3, No.2, 155-165

d'Amore. F., T. Roos, P. Widmayer (1993b): An optimal algorithm for computing a best cut of a set of hyperrectangles, International Computer Graphics Conference. Bombay

d'Amore. F., V.H. Nguyen, T. Roos. P. Widmayer (1995): On optimal cuts of hyperrectangles, Computing, Vol. 55, Springer-Verlag, 191-206

Devroye L. (1986): Lecture notes on bucket algorithms; Birkhäuser Verlag. Boston

Dröge. G., H.-J. Schek. (1993): Query-adaptive data space partitioning using variable-size storage clusters; 3rd International Symposium on Advances in Spatial Databases, Singapore, Lecture Notes in Computer Science, Vol. 692, Springer-Verlag, 337-356

Dröge, G. (1995): Eine anfrage-adaptive Partitionierungsstrategie für Raumzugriffsmethoden in Geo-Datenbanken; Dissertation ETH Zurich Nr. 11172

Edelsbrunner, H. (1982): Intersection problems in computational geometry; Dissertation. Technische Universität Graz

Enbody. R.J., H.C. Du (1988): Dynamic hashing schemes; ACM Computing Surveys, Vol. 20, No. 2, 85-113

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

MA.
Fagin, R., J. Nievergelt, N. Pippenger, H.R. Strong (1979): Extendible hashing - a fast access method for dynamic files; ACM Transactions on Database Systems, Vol. 4, No. 3, 315-344
Faloutsos, C. (1985): Multiattribute hashing using Gray codes; Proc. ACM SIGMOD International Conference on the Management of Data, Washington D.C., 227-238
Faloutsos, C. (1988): Gray codes for partial match and range queries: IEEE Transactions on Software Engineering, Vol. 14, 1381-1393
Faloutsos, C., S. Roseman (1989): Fractals for secondary key retrieval: Proc. 8th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, 247-252
Faloutsos, C., T. Sellis, N. Roussopoulos (1987): Analysis of object oriented spatial access methods; Proc. ACM SIGMOD International Conference on the Management of Data, San Francisco, 426-439
Finkel, R.A., J.L. Bentley (1974): Quad trees: A data structure for retrieval on composite keys; Acta Informatica, Vol. 4, No. 1, 1-9
Flajolet P., C. Puech (1986): Partial match retrieval of multidimensional data; Journal of the ACM, Vol. 33, No. 2, 371-407
Frank, A. (1981): Application of DBMS to land information systems; Proc. 7th International Conference on Very Large Data Bases, Cannes, 448-453
Freeston, M.W. (1987): The BANG file: a new kind of grid file; Proc. ACM SIGMOD International Conference on the Management of Data, San Francisco, 260-269
Freeston, M.W. (1989a): Advances in the design of the BANG file; Proc. 3rd International Conference on Foundations of Data Organization and Algorithms, Paris, Lecture Notes in Computer Science, Vol. 367, Springer-Verlag, Berlin, 322-338
Freeston, M.W. (1989b): A well-behaved file structure for the storage of spatial objects: Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, Lecture Notes in Computer Science, Vol. 409, Springer-Verlag, Berlin, 287-300
Freeston, M. (1995): A general solution of the n-dimensional B-tree problem. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 80-91.
Gaede, V. (1995): Optimal redundancy in spatial database systems. In Proc. 4th Int. Symp. on Spatial Databases (SSD'95).
Gaede, V., Günther, O. (1996): Multidimensional access methods; manuscript,

Humboldt-Universität Berlin, Institut für Wirtschaftsinformatik
Goodchild, M.F. (1990): Spatial information science; Proc. 4th International Symposium on Spatial Data Handling, Zurich, 3-12
Greene, D. (1989): An implementation and performance analysis of spatial data access methods; Proc. 5th International Conference on Data Engineering, Los Angeles, 606-615
Günther, O. (1988): Efficient structures for geometric data management: Lecture Notes in Computer Science, Vol. 337, Springer-Verlag, Berlin
Günther, O. (1989): The design of the cell tree: An object oriented index structure for geometric databases; Proc. 5th International Conference on Data Engineering, Los Angeles, 598-605
Günther, O. (1992a): Evaluation of spatial access methods with oversize shelves; Geographic Data Base Management Systems, ESPRIT Basic Research Series Proceedings, Springer-Verlag, 177-193
Günther, O. (1992b): Efficient computation of spatial joins; Technical Report TR-92-029, International Computer Science Institute, Berkeley
Günther, O. (1993): Efficient computation of spatial joins. In Proc. 9th IEEE Int. Conf. on Data Eng.
Günther, O., J. Bilmes (1989): The implementation of the cell tree: design alternatives and performance evaluation: GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft, Informatik-Fachberichte, Vol. 204, Springer-Verlag, Berlin, 246-265
Günther, O., J. Bilmes (1991): Tree-based access methods for spatial databases: implementation and performance evaluation: IEEE Transactions on Knowledge and Data Engineering, Vol. 3, No. 3, 342-356
Günther, O., A. Buchmann (1990): Research issues in spatial databases: IEEE CS Bulletin on Data Engineering, Vol. 13, No. 4, 35-42
Günther, o., R. Körstein, R. Müller, and P. Schmidt (1995): The MMM project: Acess to algorithms via WWW. In Proc. Third International World Wide Web Conference. URL http.//www.igd.fhg.de/www95.html.
Güting, R.H. (1989): Gral: An extensible relational database system for geometric applications; Proc. 15th International Conference on Very Large Data Bases, Amsterdam, 33-44
Guttman, A. (1984): R-trees: a dynamic index structure for spatial searching; Proc. ACM SIGMOD International Conference on the Management of Data, Boston, 47-57
Harding, E.F., D.G. Kendall (1974): Stochastic Geometry; Wiley, New York

Henrich, A. (1990): Der LSD-Baum: eine mehrdimensionale Zugriffsstruktur und ihre Einsatzmöglichkeiten in Datenbanksystemen; Dissertation, FernUniversität Hagen

Henrich, A., H.-W. Six (1991): How to split buckets in spatial data structures; Geographic Data Base Management Systems, ESPRIT Basic Research Series Proceedings, Springer-Verlag, 212-244

Henrich, A., H.-W. Six, P. Widmayer (1989a): Paging binary trees with external balancing; 15th International Workshop on Graph-Theoretic Concepts in Computer Science, Castle Rolduc, Lecture Notes in Computer Science, Vol. 411, 260-276

Henrich, A., H.-W. Six, P. Widmayer (1989b): The LSD-tree: Spatial access to multidimensional point- and non-point objects: 15th International Conference on Very Large Data Bases, Amsterdam, 45-53

Hinrichs, K.H. (1985): The grid file system: implementation and case studies of applications; Dissertation, ETH Zurich

Hutflesz, A., H.-W. Six, P. Widmayer (1988a): Globally order preserving multidimensional linear hashing; Proc. 4th International Conference on Data Engineering, Los Angeles, 572-579

Hutflesz, A., H.-W. Six, P. Widmayer (1988b): The twin grid file:A nearly space optimal index structure; Proc. International Conference on Extending Database Technology, Venice, Lecture Notes in Computer Science, Vol. 303, Springer-Verlag, Berlin, 352-363

Hutflesz, A., H.-W. Six, P. Widmayer (1988c): Twin grid files: Space optimizing access schemes; Proc. ACM SIGMOD Conference on the Management of Data, Chicago, 183-190

Hutflesz, A., H.-W. Six, P. Widmayer (1988d): Twin grid files: A performance evaluation; Proc. Workshop on Computational Geometry, CG'88, Lecture Notes in Computer Science, Vol. 333, Springer-Verlag, Berlin, 15-24

Hutflesz, A., H.-W. Six, P. Widmayer (1990): The R-file: An efficient access structure for proximity queries; Proc. 6th International Conference on Data Engineering, Los Angeles, 372-379

Hutflesz, A., P. Widmayer, C. Zimmermann (1992): Global order makes spatial access faster; Geographic Data Base Management Systems, ESPRIT Basic Research Series Proceedings, Springer-Verlag, 161-176

Iyengar, S.S., N.S.V. Rao, R.L. Kashyap, V.K. Vaishnavi (1988): Multidimensional data structures: Review and outlook; in: Advances in Computers,

ed. by Marshall C. Yovits, Academic Press, Vol. 27, 69-119
Jagadish, H.V. (1990a): Spatial search with polyhedra; Proc. 6th International Conference on Data Engineering, Los Angeles, 311-319
Jagadish, H.V. (1990b): Linear clustering of objects with multiple attributes; Proc. ACM SIGMOD International Conference on the Management of Data, Atlantic City, New Jersey, 332-342
Kamel, I. and C. Faloutsos (1994): Hilbert R-tree: An improved R-tree using fractals. In Proc. 20th Int. Conf. on Very Large Data Bases, 500-509.
Kemper, A., M. Wallrath (1987): An analysis of geometric modelling in database systems; ACM Computing Surveys, Vol. 19, No. 1, 47-91
Knuth, D. E. (1968): The art of computer programming, Vol.1: Fundamental algorithms, Addison-Wesley.
Knuth, D. E. (1973): The art of computer programming, Vol. 3: Sorting and searching, Addison-Wesley.
Kriegel, H.-P., B. Seeger (1986): Multidimensional order preserving linear hashing with partial expansions; Proc. International Conference on Database Theory, Lecture Notes in Computer Science, Vol. 243, Springer-Verlag, Berlin, 203-220
Kriegel, H.-P., B. Seeger (1987): Multidimensional dynamic quantile hashing is very efficient for non-uniform record distributions; Proc. 3rd International Conference on Data Engineering, Los Angeles, 10-17
Kriegel, H.-P., B. Seeger (1988): PLOP-Hashing: a grid file without directory; Proc. 4th International Conference on Data Engineering, Los Angeles, 369-376
Kriegel, H.-P., M. Schiwietz, R. Schneider, B. Seeger (1989a): Performance comparison of point and spatial access methods; Proc. Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, Lecture Notes in Computer Science, Vol. 409, Springer-Verlag, Berlin, 89-114
Kriegel, H.-P., B. Seeger (1989b): Multidimensional quantile hashing is very efficient for non-uniform distributions; Information Sciences, Vol. 48, 99-117
Kriegel, H.-P., T. Brinkhoff, R. Schneider (1992a): An efficient map overlay algorithm based on spatial access methods and computational geometry; Geographic Data Base Management Systems, ESPRIT Basic Research Series Proceedings, Springer-Verlag, 194-211
Kriegel, H.-P., P. Heep, S. Heep, M. Schiwietz, R. Schneider (1992b): An access method based query processor for spatial database systems; Geographic Data Base Management Systems, ESPRIT Basic Research Series Proceed-

ings. Springer-Verlag, 273-292
Krishnamurthy, R., K.-Y. Whang (1985): Multilevel grid files; IBM T.J. Watson Research Center Report, Yorktown Heights, New York
Kung, H.T., Lehman, P.L. (1980): Concurrent manipulation of binary search trees. ACM Trans. on Database Systems, Vol. 5, No. 3, 339-353
Lindenbaum, M., H. Samet (1995): A probabilistic analysis of trie-based sorting of large collections of line segments; Technical Report, University of Maryland
Litwin, W. (1980): A new tool for file and table addressing; Proc. 6th International Conference on Very Large Data Bases, Montreal, 212-223
Lo, M. and C. Ravishankar (1994): Spatial joins using seeded trees. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 209-220.
Lohmann, F., K. Neumann (1990): A geoscientific database system supporting cartography and application programming; Proc. 8th British National Conference on Databases, Pitman, London, 179-195
Lomet, D.B., B. Salzberg (1989): A robust multi-attribute search structure; Proc. 5th International Conference on Data Engineering, Los Angeles, 296-304
Lomet, D.B., B. Salzberg (1990): The hB-tree: A multiattribute indexing method with good guaranteed performance; ACM Transactions on Database Systems, Vol. 15, No. 4, 625-658
Manola, F., J.A. Orenstein (1986): Toward a general spatial data model for an object-oriented DBMS; Proc. 12th International Conference on Very Large Data Bases, Kyoto, 328-335
Matheron G. (1975): Random sets and integral geometry; Wiley, New York
Morton, G.M. (1966): A computer oriented geodetic data base and a new technique in file sequencing; IBM, Ottawa, Canada
Nelson, R.C., H. Samet (1987): A population analysis for hierarchical data structures: Proc. ACM SIGMOD International Conference on the Management of Data, San Francisco, 270-277
Nguyen, V.H., T. Roos, P. Widmayer (1993): Balanced cuts of a set of hyperrectangles; Proc. 5th Canadian Conference on Computational Geometry, Waterloo, 121-126
Nievergelt, J. (1989): 7+-2 criteria for assessing and comparing spatial data structures: Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, Lecture Notes in Computer Science, Vol. 409, Springer-Verlag, Berlin, 3-28

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

Nievergelt. J., H. Hinterberger, K.C. Sevcik (1984): The grid file: an adaptable, symmetric multikey file structure; ACM Transactions on Database Systems, Vol. 9. No. 1, 38-71

Nievergelt. J., P. Widmayer (1993): Guard files: Stabbing and intersection queries on fat spatail objects; The Computer Journal, Vol. 36, No. 2, 107-116

Nievergelt. J., P. Widmayer (1996): All space filling curves are equally efficient. Manuscript, ETH Zurich.

Ohler, T. (1992): The multiclass grid file: An access structure for multiclass range queries: Proc. 4th International Symposium on Spatial Data Handling, 260-271

Ohler, T. (1994): On the integration of non-geometric aspects into access structures for geographic information systems; Dissertation ETH Zurich Nr. 10877

Ohler, T., P. Widmayer (1994): A brief tutorial introduction to data structures for geometric databases; in: Advances in database systems: Implementations and applications, CISM courses and lectures No. 347, Springer-Verlag Wien New York, 329-351

Ohsawa. Y., M. Sakauchi (1990): A new tree type data structure with homogeneous nodes suitable for a very large spatial database; Proc. 6th International Conference on Data Engineering, Los Angeles, 296-303

Ooi. B.C. (1987): A data structure for geographic database; GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft, Informatik-Fachberichte, Vol. 136. Springer-Verlag, Berlin, 247-258

Ooi, B.C., R. Sacks-Davis, K.J. McDonell (1989): Extending a DBMS for geographic applications; Proc. 5th International Conference on Data Engineering, Los Angeles, 590-597

Orenstein. J.A. (1986): Spatial query processing in an object-oriented database system; Proc. ACM SIGMOD International Conference on the Management of Data, 326-336

Orenstein. J.A. (1989): Redundancy in spatial databases; Proc. ACM SIGMOD International Conference on the Management of Data, Portland, 294-305

Orenstein. J.A. (1990): A comparison of spatial query processing techniques

for native and parameter spaces; Proc. ACM SIGMOD International Conference on the Management of Data, Atlantic City, New Jersey, 343-352
Orenstein, J.A., T.H. Merrett (1984): A class of data structures for associative searching; Proc. 3rd ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, Waterloo, 181-190
Otoo, E.J. (1986): Balanced multidimensional extendible hash tree; Proc. 5th ACM SIGACT-SIGMOD International Symposium on Principles of Database Systems, Cambridge, Massachusetts, 100-113
Otoo, E.J. (1990): An adaptive symmetric multidimensional data structure for spatial searching; Proc. 4th International Symposium on Spatial Data Handling, Zurich, 1003-1015
Ouksel, M. (1985): The interpolation-based grid file: Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Portland, 20-27
Overmars, M. (1981): Dynamization of order decomposable set problems, J. of Algorithms, Vol. 2, 245-260
Overmars, M. (1983): The design of dynamic data structures, Proefschrift, Rijksuniversiteit Utrecht
Ozkarahan, E.A., M. Ouksel (1985): Dynamic and order preserving data partitioning for database machines; Proc. 11th International Conference on Very Large Data Bases, Stockholm, 358-368
Pagel, B.-U. (1995): Analyse und Optimierung von Indexstrukturen in Geo-Datenbanksystemen; Dissertation, FernUniversität Hagen, Germany
Pagel, B.-U., H.-W. Six, H. Toben (1993a): The transformation technique for spatial objects revisited; 3rd International Symposium on Advances in Spatial Databases, Singapore, Lecture Notes in Computer Science, Vol. 692, Springer-Verlag, 73-88
Pagel, B.-U., H.-W. Six, H. Toben, P. Widmayer (1993b): Towards an analysis of range query performance in spatial data structures; 12th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Washington, D.C., 214-221
Preparata, F.P., M.I. Shamos (1985): Computational Geometry: An Introduction, Springer Verlag, Berlin, Heidelberg, New York
Ramamohanarao, K., R. Sacks-Davis (1984): Recursive linear hashing; ACM Transactions on Database Systems, Vol. 9, No. 3, 369-391
Regnier, M. (1985): Analysis of grid file algorithms: BIT Vol. 25, 335-357
Robinson, J.T. (1981): The K-D-B-tree: a search structure for large multidimensional dynamic indexes; Proc. ACM SIGMOD International Conference

on the Management of Data, Ann Arbor, 10-18
Roman, G.-C. (1990): Formal specification of geographic data processing requirements: IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 4, 370-380
Rottke T., H.-W. Six, P. Widmayer (1987): On the analysis of grid structures for spatial objects of non-zero size; International Workshop on Graph-Theoretic Concepts in Computer Science, Staffelstein, Lecture Notes in Computer Science, Vol. 314, Springer-Verlag, Berlin, 94-105
Roussopoulos, N., D. Leifker (1985): Direct spatial search on pictorial databases using packed R-trees; Proc. ACM SIGMOD International Conference on the Management of Data, Austin, 17-31
Sagan, H. (1994): Space-Filling Curves. Berlin/Heidelberg/New York: Springer-Verlag.
Samet, H. (1988): Hierarchical representations of collections of small rectangles; ACM Computing Surveys, Vol. 20, No. 4, 271-309
Samet, H. (1990a): The design and analysis of spatial data structures; Addison-Wesley, Reading
Samet, H. (1990b): Applications of spatial data structures; Addison-Wesley, Reading
Santalo, L.A. (1976): Integral geometry and geometric probability; Addison-Wesley, Reading
Schek, H.-J., W. Waterfeld (1986): A database kernel system for geoscientific applications; Proc. 2nd International Symposium on Spatial Data Handling, Seattle, 273-288
Schek, H.-J., A. Wolf (1993): From extensible databases to interoperability between multiple databases and GIS applications; 3rd International Symposium on Advances in Spatial Databases, Singapore, Lecture Notes in Computer Science, Vol. 692, Springer-Verlag, 207-238
Schiwietz, M., H.-P. Kriegel (1993): Query processing of spatial objects: Complexity versus redundancy, 3rd International Symposium on Advances in Spatial Databases, Singapore, Lecture Notes in Computer Science, Vol. 692, Springer-Verlag, 377-396
Seeger, B. (1989): Entwurf und Implementierung mehrdimensionaler Zugriffsstrukturen: Dissertation, Universität Bremen
Seeger, B., H.-P. Kriegel (1988): Techniques for design and implementation of efficient spatial access methods; Proc. 14th International Conference on Very Large Data Bases, Los Angeles, 360-371

Seeger, B., H.-P. Kriegel (1990): The buddy-tree: An efficient and robust access method for spatial data base systems; Proc. 16th International Conference on Very Large Data Bases, Brisbane, 590-601

Sellis T., N. Roussopoulos, C. Faloutsos (1987): The R+-tree:A dynamic index for multi-dimensional objects; Proc. 13th International Conference on Very Large Data Bases, Brighton, 507-518

Shaffer, C.A., H. Samet (1990): Set operations for unaligned linear quadtrees; Computer Vision, Graphics, and Image Processing, Vol. 50, No. 1, 29-49

Shaffer, C.A., H. Samet, R.C. Nelson (1990b): QUILT: A geographic information system based on quadtrees; International Journal of Geographical Information Systems, Vol. 4, No. 2, 103-131

Six H.-W., P. Widmayer (1988): Spatial searching in geometric databases; Proc. 4th International Conference on Data Engineering, Los Angeles, 496-503

Six H.-W., P. Widmayer (1992): Spatial access structures for geometric databases: in: Data Structures and Efficient Algorithms, Ed. B. Monien, T. Ottmann, Final Report on the DFG Special Joint Initiative, Lecture Notes in Computer Science, Vol. 594, 214-232.

Smith, T.R., P. Gao (1990): Experimental performance evaluations on spatial access methods; Proc. 4th International Symposium on Spatial Data Handling, Zurich, 991-1002

Stoyan, D., W.S. Kendall, J. Mecke (1987): Stochastic geometry and its applications: Wiley, New York, 1987.

Subramanian, S., S. Ramaswamy (1995): The P-range tree: A new data structure for range searching in secondary memory: in Proc. Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 378-387

Tamminen, M. (1982): The extendible cell method for closest point problems; BIT, Vol. 22, 27-41

Tamminen, M. (1984): Metric data structures - an overview; Helsinki University of Technology, Laboratory of Information Processing Science, Report HTKK-TKO-A25

Tamminen, M. (1985): On search by address computation: BIT Vol. 25, 135-147

Tropf, H., H. Herzog (1981): Multidimensional range search in dynamically balanced trees; Angewandte Informatik, Vol. 2, 71-77

van Oosterom, P. (1990): Reactive data structures for geographic informa-

tion systems; Proefschrift, Rijksuniversiteit Leiden
van Kreveld, M.J. (1992): New results on data structures in computational geometry; Prafschrift, Rijksuniversiteit Utrecht
van Oosterom, P., E. Claassen (1990): Orientation insensitive indexing methods for geometric objects; Proc. 4th International Symposium on Spatial Data Handling, Zurich, 1016-1029
van Oosterom, P. (1990): Reactive data structures for geographic information systems; Proefschrift, Rijksuniversiteit Leiden Wang, J.-H., T.-S. Yuen, D.H.-C. Du (1987): On multiple random access and physical data placement in dynamic files; IEEE Transactions on Software.Engineering, Vol. 13, No. 8, 977-987
Waterfeld, W. (1991): Eine erweiterbare Speicher- und Zugriffskomponente für geowissenschaftliche Datenbanksysteme; Dissertation, Fachbereich Informatik, Technische Hochschule Darmstadt, Germany
Widmayer, P. (1991): Datenstrukturen für Geodatenbanken; in: Entwicklungstendenzen bei Datenbank-Systemen, ed. G. Vossen, K.-U. Witt, Oldenbourg Verlag München Wien, 317-361
Willard, D.E., (1978): Balanced forests of k-d* trees as a dynamic data structure, TR-23-78, Aiken Computation Lab., Harvard University.
Xu, X., J. Han, W. Lu (1990): RT-tree: an improved R-tree index structure for spatiotemporal databases; Proc. 4th International Symposium on Spatial Data Handling, Zurich, 1040-1049

DISCUSSION

Rapporteur: Michael Elphick

Professor Randell remarked that in his introduction, the speaker had omitted the whole history of the development of punched-card processing techniques.

Later, Dr Goldberg asked how well the data-structures described by the speaker coped with the problems posed by animation, virtual reality and the like, where the need for coherent movement imposed rather strong requirements. Professor Nievergelt said that this was an area about which he had little to say; however, he would suggest that one must have some kind of "coarse" object model, for representing such objects as a train moving at constant speed. His data-structures certainly didn't solve all problems.

Professor Randell then referred to the problems faced by mapping organisations (such as the British Ordnance Survey), who had initially simply inserted the unstructured data previously held on paper into their early databases. Professor Nievergelt responded that in one sense these were simpler problems, in that the data was largely static. However, they had to deal with many more types of relationship than the purely geometrical. He also noted that in typical GIS applications, the total amount of data was not staggeringly large; perhaps of the order of several hundred thousand basic items.

As a separate point, Professor Randell observed that in talking to other colleagues, there seemed to be two distinct types of GIS: one principally concerned with representing and visualising physical data, and the other with the correlating of different kinds of data (perhaps relating ZIP codes with administrative divisions).

Professor Nievergelt felt that this was to go way beyond the problems of structuring spatial data, and dealt with areas about which he had no particular knowledge. His talk had dealt with a restricted area, and this might well be only a minor part of the whole problem in such cases.