# Multidimensional Data Structures: Review and Outlook

S. SITHARAMA IYENGAR
N. S. V. RAO

*Department of Computer Science*
*Louisiana State University*
*Baton Rouge, Louisiana 70803*

R. L. KASHYAP[†]

*Department of Electrical Engineering*
*Purdue University*
*West Lafayette, Indiana 47907*

V. K. VAISHNAVI

*Department of Computer Information Systems*
*Georgia State University*
*University Plaza*
*Atlanta, Georgia 30303*

# 1.  Introduction

The design and analysis of data structures and algorithms is an important area of computer science. In recent years there have been several significant advances in this field, ranging from the binary search algorithm to the startling discovery of the polynomial algorithm for the linear programming (LP) problem. These results have produced efficient and optimal solutions to many real-life problems. As a consequence, much interest has been generated in the field of data structures and algorithms, which have blossomed into the forefront of computer science research.

Searching an ordered collection of data is a standard paradigm in the area of algorithms. The problem of search in a single dimension has been handled traditionally using the data structures of binary trees, multiway trees, hash tables, etc. In many applications arising in information retrieval, computational geometry, robotics, and statistics, the central problem is, however, one of determining the points falling in specified range limits in different dimensions. These problems have spurred an enormous interest in the design and analysis of efficient multidimensional data structures. In addition, the multidimensional data structures address themselves to a larger class of problems than single-dimensional data structures. See, for details, Horowitz and Sahni (1976). But the multidimensional data structures are not natural extensions of single-dimensional data structures. Instead, they are more complex and problem oriented. For example, consider the balanced binary tree data structure. The cost of constructing such a data structure is $O(N \log N)$, where $N$ is the number of data points in the structure. The corresponding storage cost is $O(N)$. This data structure can answer complete match and range queries in $O(\log N)$ and $O(\log N + K_1)$ time respectively,

where $K_1$ is the number of qualified data points that satisfy the query. There are no multidimensional data structures with similar performace.

A number of multidimensional data structures such as $k$-d trees, quadtrees, range trees, polygon trees, quintary trees, multiple attribute tree (MAT), balanced multidimensional and weighted trees, etc. have been studied recently for range and related search problems.

A $k$-dimensional search tree is a $k$-dimensional generalization of a search tree. It can be used for items that are $k$-tuples. When the ($k$-dimensional) data items have weights (positive real numbers) associated with them, the data items are called weighted items. A $k$-dimensional tree for such weighted items is a weighted tree of $k$-dimensions. In this paper we will also explore the relationship between weighted trees and balanced multidimensional trees.

## 1.1   Organization and Scope of the Paper

The focus of the initial part of this paper is to present and analyze simple and efficient balanced multidimensional tree structures that are useful for real time applications. The usefulness of such structures for very large data and concurrent environments is an important concern of future research in multidimensional data structures. A major theme of this part of the paper is the presentation of a paradigm for generalizing a balanced tree structure to multidimensions, along with a set of strategies for supporting structure violations in the generalized structures. The later part of the paper elaborates on various aspects of multiple attribute trees.

Section 2 presents an overview on performance of various multidimensional data structure. Section 3 describes balanced multidimensional and weighted trees as well as their applications in physical database organization, information retrieval, and computational geometry problems. Section 4 reviews the literature for generating a balancing concept for multidimensional trees. Section 5 describes a paradigm that models the available balanced and semi-balanced multidimensional and weighted structures. These paradigms are expected to guide future research in this area. Section 6 describes multiple attribute tree structures for partial match, complete match, and range queries. Also discussed is a comparison of the performance of the MAT structure with other multidimensional structures. Section 7 discusses paradigms for modeling the salient treatment of multiple attribute trees. Finally, Section 8 presents some conclusions.

## 2.   What is a Multidimensional Data Structure?

A datum or data element is said to be *single-dimensional* if it is characterized by a single unique attribute or key value. Examples for single-dimensional

data structures are balanced binary trees, B-trees, AVL-trees, hash tables, multiway trees, etc.

A data element is said to be *multidimensional* if it is characterized by a number of attributes. A multidimensional data element specified by $k$ attributes can be imagined to be a "point" in a $k$-dimensional space. Examples for multidimensional data structures are $k$-d trees, quadtrees, multiple attribute trees (MAT), multidimensional B-trees, balanced multidimensional and weighted trees, etc. A multidimensional problem calls for a efficient data structure that organizes these points in such a way as to result in efficient access. The problem of retrieving all the records satisfying a query involving a multiplicity of attributes, known as *associative retrieval*, is a major issue in physical database organization. The factors which affect the performance of physical data organization are logical data characteristics, query complexity and device parameters, and it has been demonstrated that no single index structure can be optimal under all circumstances. For more on this see Kriegel (1981).

The performance of a multidimensional data structure is measured in terms of the following three quantities (We assume $N$ points, each having $k$ attributes, in our discussions):

- $P(N)$, *preprocessing cost*, which is the time taken for constructing the data structure;
- $S(N)$, *storage cost*, which is the amount of storage needed to store the data structure;
- $Q(N)$, *access cost*, which is the time taken for accessing the data structure as per the requirement. This can be query cost when answering a query about the data. In general, the cost embedded in this can be attributed to insertion, deletion or any other modification.

A *static* data structure is constructed only once initially. Queries can be answered in this data structure, but no insertions and deletions are supported. In a *dynamic* data structure, the insertions and deletions are supported dynamically. As will be shown in the later sections, maintenance of the data structure in the optimal form of a static one is a very difficult task. This is mainly because the insertions and deletions will, in general, destroy the balanced nature of the data structure that can be obtained in the static case. The dynamic data structures are indispensable for a large number of applications.

We now present some multidimensional data structures that arise frequently in several applications. The vast amount of literature and the varied nature of the proposed data structures prohibits an exhaustive treatment here. Therefore, this paper covers, in detail, balanced multidimensional and

weighted trees, multiple attribute trees (MAT), and to a lesser extent other multidimensional data structures. Paradigms for modeling balanced multidimensional and weighted trees as well as MAT are presented in detail in section 7.

## 2.1   A Classification of Multidimensional Search Problems

One important class of multidimensional problems is the *search problem*. Let $T_1$, $T_2$, $T_3$ be sets of queries, elements, and answers respectively. A searching problem $P$ on $T_1$, $T_2$, $T_3$ is a function

$$P : T_1 \times 2^{T_2} \quad \rightarrow \quad T_3.$$

The searching problem $P$ takes a query object $t_1 \in T_1$ and a domain set $t_2 \in 2^{T_2}$ and produces an answer $t_3 \in T_3$. By assigning different properties to $T_1$, $T_2$, and $T_3$, the different classes of searching problems are obtained.

In a *member searching* problem $T_1 = T_2$ and $T_3 = \{\text{yes, no}\}$. The multidimensional searching problem can be converted to a single-dimensional one, by concatenating all attributes to form a single key. Hence, the standard approaches for single-dimensional cases, such as in Aho, Hopcroft, Ullman (1974) and Knuth (1974) can be applied to obtain $Q(N) = O(k \log N)$, $P(N) = O(kN \log N)$, $S(N) = O(N)$. The same complexity estimates can be obtained for searching for the $r^{\text{th}}$ element or for finding the rank of an element in an ordered set as in Overmars (1984).

Another important searching problem is the *range search*. In this problem, $T_2$ contains points from $k$-dimensional space, and each $q \in T_1$ specifies a range for each dimension. $T_3$ is a subset of $T_2$ and contains the points that satisfy $q$. The point quadtree proposed by Finkel and Bentley (1974) and Lee and Wong (1977) has the complexity estimates of $Q(N) = O(N^{1 - 1/k} + K_1)$, $P(N) = O(N \log N)$, $S(N) = O(N)$, where $K_1$ is the number of points that satisfy the query. A point quadtree for a given set of points is a $2^k$-ary tree, which recursively divides the space into $2^k$ quadrants. Although the worst-case time is not very impressive, in practice the quadtree is shown to be efficient in many practical problems. Details can be found in Finkel and Bentley (1974), Bentley and Stanat (1975) and Alt *et al.* (1981).

An important data structure for solving the range searching problem is called the *k-d tree*. This data structure is introduced by Bentley (1975). A *k*-d tree is a type of binary tree built as follows: A data point is taken as the root node; the data space is divided into two subsets based on the first attribute value; each of these sets is represented as a subtree of the root; each subtree is split based on the second attribute value. This process is carried out recursively on the subsets. After splitting with respect to the $k^{\text{th}}$ attribute, the first attribute is considered again. Bentley (1975) and Lee and Wong (1977)

solve the range searching problem with the following performance measures: $Q(N) = O(N^{1 - 1/k} + K_1)$, $P(N) = O(N \log N)$, $S(N) = O(N)$. Mehlhorn (1984b) discusses dd-trees which are very similar to $k$-d trees. At each step the data space is divided into three subsets, in contrast with two for the $k$-d tree. The first subset contains all the points whose corresponding attribute value is less than that of the root. The second subset contains all the points that have the same attribute value as the root. The third subset contains all the points that have the attribute value greater than that of the root. Both structures have similar characteristics.

There is another data structure for the range query which has a better worst-case query cost at the expense of high preprocessing and storage costs. This data structure has been independently proposed by various authors, and each differs from the other only slightly in the structural aspects. The performance measures of this data structure are $Q(N) = O(\log^k N + K_1)$, $P(N) = O(N \log^{k-1} N)$, $S(N) = O(N \log^{k-1} N)$. The basic data structure that underlies all these versions, referred to here as a *range tree*, can be defined as follows: For the single dimensional case the range tree reduces to a balanced binary leaf search tree in which the points are stored in the leaf nodes. Moreover, the leaves are linked in that order. To perform a range query, we search for the limits. Then, the points that lie within the range limits are retrieved using the linked list structure. A $k$-dimensional range tree consists of a balanced binary leaf search tree $T$ in which points are stored and ordered according to the first attribute. To each internal node $\alpha$, a $(k - 1)$-dimensional range tree $T_\alpha$ is associated. This subtree rooted at $\alpha$ takes into account only the second to $k^{\text{th}}$ attributes for all points of the data. The basic underlying data structure here is a single-dimensional binary search tree. Conceptually, one can think of using the other single dimensional data structures such as the AVL-tree, B-tree, 2-3 tree, BB[$\alpha$] tree, etc, as the underlying data structure. The order estimates for the range query remain the same, but the exact number of steps for a particular query differs. Willard (1979b) uses B-tree, and the corresponding multidimensional data structure is called the *super B-tree*. Lueker (1978) uses BB[$\alpha$] tree as the underlying data structure, and the multidimensional version is called the *d-fold tree*. The *quintary tree* of Lee and Wong (1980) also falls into this type. They use a five-way tree as the underlying structure. The other single-dimensional data structures can also be used to obtain new multidimensional data structures. However, such extensions do not seem to promise more efficient structures than the existing ones. Willard (1985) and Hart (1981) tighten the query cost to $Q(N) = O(\log^{k-1} N + K_1)$ using a clever technique. Willard (1985) refers to this data structure as a *k-fold tree*. A number of other data structures such as *overlapping k-ranges, non-overlapping k-ranges*, etc. are proposed. These structures illustrate the different trade-offs between the query time, preprocessing time, and the storage

cost. The performance measures for the overlapping $k$-ranges are: $Q(N) = O(\log N + K_1)$, $P(N) = O(N^{1+\epsilon})$, and $S(N) = O(N^{1+\epsilon})$, for $\epsilon > 0$. The non-overlapping $k$-ranges have performance measures as follows: $Q(N) = O(N^\epsilon + K_1)$, $P(N) = O(N \log N)$, and $S(N) = O(N)$. The reduction in one of the costs is reflected as an increase in the other costs. Bentley and Friedman (1979) gives an excellent survey on the data structures for the range search. More on the range and related issues can be found in Overmars (1984), Mehlhorn (1984b), Willard (1985) and Willard and Lueker (1985).

Another often encountered searching problem is the *nearest neighbor searching* problem. This problem consists of finding a point in $T_2$ that is the nearest to a query point $x \in T_1$, with respect to some metric. This problem frequently arises in database systems. This problem is shown to have the solution given by $Q(N) = O(\log N)$, $P(N) = O(N \log N)$, and $S(N) = O(N)$ for the two-dimensional case. The details can be found in Overmars (1984), Shamos and Hoey (1975), Shamos (1978) and Kirkpatrick (1983).

The multidimensional data structures such as the $k$-d trees, quadtrees, range trees, etc., are comparison-based data structures. There is another class of multidimensional tree structures based on the digital search of nodes in the tree, (Knuth (1974)). The nodes in the comparison-based structures are formed based on the result of a comparison between two elements. In the digital-based tree structures, the nodes are formed based on the values of the elements. The class of multidimensional tree structures based on the digital techniques includes *modified doubly chained trees* of Cardenas and Sagamang (1974), *binary search tree complexes* (BST-complex) of Lien *et al.* (1975), *multidimensional B-tree* (MDBT) of Ouksel and Scheuermann (1981), *kB-trees* of Gotlieb, and Gotlieb (1978), the $kB^+$-*tree* of Kriegel and Vaishnavi (1981a), and the *multiple attribute trees* (MAT) of Kashyap *et al.* (1977), Gopalakrishna and Veni Madhavan (1980), and Rao *et al.* (1984). The main difference between the comparison and the digital multidimensional tree data structures lies in the way the data is processed in building the data structure. For comparison-based methods the multidimensional space is recursively divided into smaller regions of the same dimensionality. In the digital-based structures the dimensionality is recursively reduced by one. This distinction is not applicable in the single-dimensional case, but is a vital factor in deciding the performance of multidimensional data structures. In the comparison data structures the search is recursively carried out in the smaller regions of the same dimensionality, whereas in the digital data structures the search is recursively carried out in the space of reduced dimensionality. This means that there could be applications that are naturally suited to one of the two types.

The area of computational geometry is rich in multidimensional problems and data structures. This emerging area has found many applications in various disciplines such as pattern recognition, operations research, computer

graphics, robotics, etc. Computational geometry principally deals with the computational complexity of geometric problems. The five major areas in this field are convex hull, intersections, searching, proximity finding, and combinatorial optimization. The search problems discussed in this paper are related to the search and proximity problems of computational geometry. Several generic algorithms, such as divide and conquer, incremental construction, plane sweep, locus, geometric transformations etc. have been developed for various problems. For details on computational geometry problems see Mehlhorn (1984b), Edelsbrunner (1983, 1984), Overmars (1984), Preparata and Shamos (1985), and Shamos (1978).

In the area of information retrieval and management, a number of multidimensional data structures have been studied. The *multidimensional trees* and the *multidimensional directory* of Orenstein (1982) are some of the important ones. The structures such as *inverted lists* and *multi lists* of Liou and Yao (1977) are examples of extensions of the concepts of lists to multiple dimensions. The *grid file* of Nievergelt *et al.* (1984) is another multidimensional data structure that supports multikey access. This structure is proved to be superior to conventional structures such as inverted files and multi lists in terms of the adaptation to the dynamic environments. Unlike the earlier mentioned data structures, the grid file deals with the storage of the directory on the secondary memory. There is also a pedigree of data structures for the management of the directory on the secondary memory. The generalization of hash functions by Rothnie *et al.* (1974) describes a multikey hash file structure. But finding a hash function that optimizes the query, insertion, deletion, collision resolution, and storage costs is very difficult (see Lum (1970)) and problem dependent. Lum (1970) describes *combined indices* that are formed by concatenating several attribute values. The *compressed bit maps* are based on the bit-encoded inverted files. The *transposed file* organization of Batory (1979) and Barnes and Collens (1973) also makes use of compression techniques in providing multikey access. The generalization of *tries* to support multikey access can be found in Orenstein (1982) and Tamminen (1982). Casey (1973) describes a complex-tree-based multikey access structure, which uses *superimposed* coding to characterize the records below the nodes of the tree. This approach is complex, and a more practical approach based on similar concepts can be found in Pealtz *et al.* (1980). The multidimensional directories suggested by Liou and Yao (1977) provide an efficient way of multikey access, using priority ordering of the attributes. The *interpolation*-based index maintenance of Burkhard (1983) is a multidimensional extension of the linear hashing of Litwin (1980), and is related to the extendible hashing of Fagin *et al.* (1979). These multikey access methods for the data stored in the secondary memory provide efficient solutions for many practical applications. Since many of these approaches rest on identifying a suitable hash function, they are not very amenable to rigorous theoretical analysis.

One important aspect of the multidimensional data structures is *dynamization*. Ideally, the term "dynamization" refers to the process of maintaining the data structure to ensure the performance of the static data structure under the dynamic environment of insertions and deletions of records. Normally, the dynamization of the multidimensional data structures is much more involved than that in single dimensional data structures. This problem becomes further complicated when the data directory resides on the secondary memory. This is because, in many cases, the insertions and deletions destroy the clustering of the data into secondary memory pages. Many data structures are dynamic when implemented in memory that supports random access. But the considerations become markedly different when the data structure is stored on the secondary memory. Again, the requirements of dynamization vary considerably depending on the application. The general framework for dynamization is introduced by Bentley (1979a, 1980) for a certain class of problems called *decomposable* problems. In Bentley and Saxe (1980) the general concepts of static-to-dynamic transformation of data structures are discussed. The notions of *local, partial,* and *global* rebuilding of data structures are discussed by Overmars (1984) for the purpose of dynamization. The varied nature of the dynamization makes the generalized treatment difficult. For details of a simple dynamization method see Rao, Vaishnavi and Iyengar (1987).

## 3.   Balanced Multidimensional and Weighted Trees

### 3.1   Background

Vaishnavi (1984, 1987) recently initiated a new direction of research for developing balanced self-organizing and multidimensional search trees that support various operations through algorithms that are relatively simple to understand and implement. This research involves generalizing balanced search trees using a new paradigm for supporting a structure violation by an EQSON-subtree.

In this section, we will discuss certain basic concepts and issues as well as review related real-time applications for balanced multidimensional and weighted trees. The paradigms described in this section provide a methodology for generalizing balanced search tree structures to multidimensions.

### 3.1.1   k-dimensional (Search) Trees

A *k-dimensional (search) tree*, as illustrated in Fig. 1, is a natural data structure for *k*-dimensional data. It is related to TRIEs developed by Fredkin (1960), digital search trees by Knuth (1974), and lexicographic trees by Sleator and Tarjan (1985). It can also be interpreted as a *k-dimensional* generalization
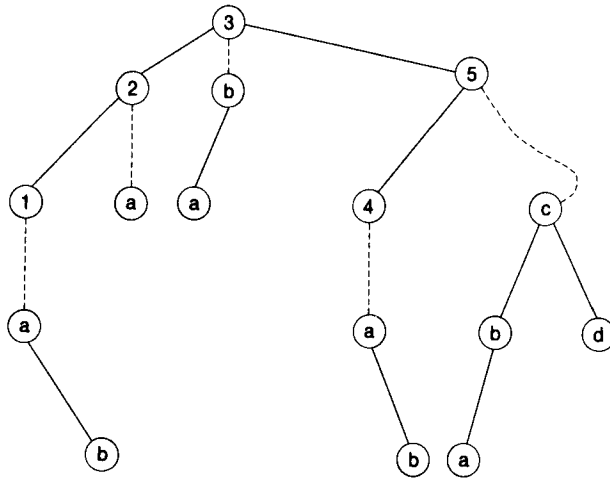
FIG. 1.   A 2-dimensional search tree for the following set of 2-dimensional data items: $\{(1, a),$ $(1, b), (2, a), (3, a), (3, b), (4, a), (4, b), (5, a), (5, b), (5, c), (5, d)\}$. For the data item $(1, a)$, the first attribute is "a" and the second attribute is "1".

of a (one-dimensional) search tree. In such a (ternary) tree, as we have the following (Bentley and Saxe (1979): The root stores $x_k$, a $k$-dimensional key, the $k^{th}$ attribute of a data item. The left structure of the root is a $k$-dimensional tree for those data items whose $k^{th}$ attributes are less than $x_k$. The right subtree of the root is a $k$-dimensional tree for those data items whose $k^{th}$ attributes are greater than $x_k$. The middle subtree of the root, an *EQSON-subtree*, is a $(k - 1)$-dimensional tree for the $(k - 1)$-dimensional projection data items obtained by omitting $x_k$, the identical $k^{th}$ attribute of data items. A $k$-dimensional (binary) leaf tree (useful for sequential processing) and a $k$-dimensional multiway tree (useful for very large data) can be similarly defined. The worst-case performance per operation of a $k$-dimensional tree is rather poor unless its structure is properly constrained. A *balanced k-dimensional (search) tree* is a constrained tree structure such that its height is at most logarithmic, i.e., $O(\log n + k)$, which is optimal as shown by Bentley and Saxe (1979) and Hirschberg (1980), and is a flexible enough tree structure such that it can be updated in logarithmic worst-case time per operation.

### 3.1.2   Weighted Trees

When the ($k$-dimensional) data items have *weights* (positive real numbers) associated with them, the data items are called *weighted items*. A $k$-dimensional tree for such weighted items is a *weighted (k-dimensional) tree*. An obvious interpretation of the weight of a data item is that it represents the

probability with which the item is accessed. A weighted tree, with weights representing access probabilities, that dynamically adapts itself to the (possibly non-uniform) access pattern is a *self-organizing tree*. There can be other interpretations of weights too, such as the one used in the application of the D-trees developed by Mehlhorn (1984a) for the dynamization of interval trees (see Edelsbrunner (1980a), McCreight (1980)) and segment trees (see Bentley (1977)).

In a self-organizing tree, two additional operations, namely *promote* (increase the weight of an item by a certain amount) and *demote* (decrease the weight of an item by a certain amount), become relevant. The best that can be expected from a self-organizing (*k*-dimensional) tree is that its worst-case access time is "logarithmic," i.e., $O(\log W/w_i + k)$, where $w_i$ is the weight of item $i$ at a certain instant of time and $W$ is the sum of the weights of all items at that time. For a broader treatment on this see Gueting and Kriegel (1981) and Mehlhorn (1984a). The worst-case time for other operations can at best be logarithmic. A *balanced weighted (k-dimensional) tree* can be accessed and updated in logarithmic worst-case time per operation.

## 3.2   Multidimensional Trees and Weighted Trees

Weighted trees and multidimensional trees are related to each other. One can start from an efficient weighted one-dimensional tree structure and use it for constructing efficient multidimensional or even efficient weighted multidimensional tree structures by using it appropriately for implementing nodes of TRIEs developed by Fredkin (1960). Alternatively, one can start from an efficient $(k + 1)$-dimensional tree structure and use it for constructing an efficient weighted $k$-dimensional tree structure by letting the lowest level subtrees used as virtual subtrees representing weights of items (see Gueting and Kriegel (1981)).

### 3.2.1   *Worst-Case vs. Amortized Behavior*

Many applications require the use of a sequence of operations. In such cases, it is not important that the *worst-case* time per operation be small; it suffices to have the operation times *amortized* over a worst-case sequence of operations to be small (see Bent *et al.* (1980) and Huddleston and Mehlhorn (1982)). The amortized time per operation is also a good measure of the average performance of a data structure as shown by Bentley and McGeoch (1985). There are situations, however, where it is important that the worst-case time per operation be small. A real-time application is an example of such a situation. Also, a small worst-case time per operation sometimes gets translated into a still smaller amortized time per operation. Additionally, a

good worst-case performance of a data structure that is the basis for a more complex data structure can sometimes be important for a good amortized behavior of the latter (see, for example, dynamic interval and segment trees based on D-trees Mehlhorn (1984b)). Here, we will mainly focus on balanced multidimensional and weighted trees, which are worst-case efficient data structures. The Splay tree, presented by Sleator and Tarjan (1983a), is an elegant data structure which guarantees amortized-case efficient behavior for access as well as update operations.

## 3.3  Applications

Balanced multidimensional and weighted trees have applications in such areas as (a) physical database organization, (b) information retrieval, (c) self-organizing file structures, and (d) computational geometry. They may be used as independent data structures (see Bent *et al*. (1985), Gueting and Kriegel (1980) and Gueting and Kriegel (1981)), or as underlying schemes for complex data structures (Mehlhorn (1984b), Sleator and Tarjan (1983b), or may be used in a suitably enhanced form in complex applications (Kriegel (1984), Kriegel and Vaishnavi (1981c), and Scheuermann and Ouksel (1982)). Let us elaborate on these applications.

### 3.3.1  *Physical Database Organization*

The problem of retrieving all the records satisfying a query involving a multiplicity of attributes, known as multikey retrieval or associative retrieval, is a major concern in physical database organization (Kriegel (1984)). The expanding use of database systems for executive decision-making demands a physical organization that can support fast responses to associative queries as well as efficient updates. The available file structures (e.g., the inverted file of Knuth (1974), the grid file of Nievergelt *et al*. (1984), the MDBT organization of Scheuermann and Ouksel (1982), the $k$B-trees of Gueting and Kriegel (1980) and Kriegel (1984)), suffer from certain deficiencies, especially if the database is highly dynamic, and associative queries such as range queries and partial-range queries are important.

A balanced multidimensional multiway tree structure, suitably enhanced with additional pointers as in the MDBT organization by Scheuermann and Ouksel (1982), can serve as a dynamic file organization for associative retrieval. The performance of $k$B-trees of Gueting and Kriegel (1980) and Kriegel (1984), a balanced multidimensional multiway tree structure (further discussed in Section 5), has been compared (see Kriegel (1984)) with the inverted file (Knuth (1974)), the grid file (Nievergelt et al. (1973)), and the MDBT-organization (Scheuermann and Ouksel (1982)). The inverted file has

been found to have the worst performance; while the grid file has been found to have the best performance in certain respects, $k$B-trees have been found to have the best performance in certain other respects.

### 3.3.2   Information Retrieval

A balanced multidimensional (search) tree can serve as a space-efficient implementation of a TRIE developed by Fredkin (1960) or digital search tree of Knuth (1974), a data structure commonly used in information retrieval (see Fig. 2). If each node of a TRIE for a set of English words is implemented as an array, then a word can be retrieved, inserted, or deleted in time proportional to the length of the word. But this implementation is quite wasteful in storage. One the other hand, an implementation based on a balanced multidimensional tree structure occupies storage proportional to the total number of characters in words and supports retrieval and update in time proportional to the length of the word plus log $n$ where $n$ is the total number of words. This performance is optimal as demonstrated by Hirschberg (1980) if comparison between characters is a unit operation.
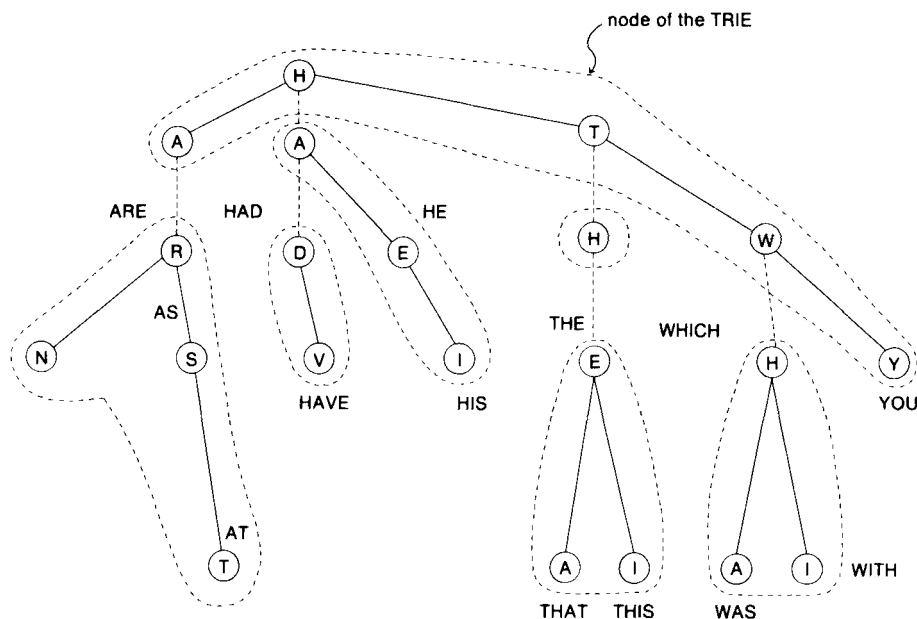


FIG. 2.   A TRIE for the English words {AN, ARE, AS, AT, HAD, HAVE, HE, HIS, THAT, THE, THIS, WAS, WHICH, WITH, YOU}.

### 3.3.3   Self-Organizing File Structures

If the weights of data items are interpreted as frequencies with which they are accessed, then a balanced weighted tree structure is a dynamic self-organizing file structure which adapts itself to usage. It stays in a nearly optimal form with respect to the frequencies of access, without needing *a-priori* information on these frequencies and without using counters, etc., for recording them. A balanced weighted multidimensional tree structure can thus be used as a self-organizing file structure in databases (see Kriegel and Vaishnavi (1981c)) or in information retrieval.

### 3.3.4   Computational Geometry

Computational geometry deals with algorithmic aspects of geometrical problems. The computational problems arising in two- and three-dimensional computer graphics and in VLSI design have generated much interest in this field.

Interval trees (Edelsbrunner (1980a), McCreight (1980)) and segment trees (Bentley (1977)) are some of the new data structures that have been discovered for developing efficient algorithms for problems related to computational geometry. They are, however, static data structures; their underlying data structure is a completely balanced binary tree. They can, however, be made dynamic by using a suitable balanced weighted tree structure as the underlying data structure and by interpreting the weights appropriately. D-trees (Mehlhorn (1979)) (further discussed in Section 4), a balanced weighted tree structure that has a rather strong bound on its update time, has been successfully used for the dynamization of interval and segment trees. However, D-trees are not convenient to use because they require complex restructuring operations in their update algorithms.

## 4.   Balanced Multidimensional and Weighted Trees: A Close Look at the Literature

Besides balanced multidimensional and weighted tree structures, we include in this review *semi-balanced* multidimensional and weighted tree structures — tree structures that can be accessed in logarithmic time in the worst-case but can be updated in logarithmic time only in the amortized case. Two approaches have been used for discovering such data structures:

(a) extending a balanced tree structure, and
(b) generalizing a balancing concept.

The data structures developed, even using the same approach, differ in their basic strengths, performance, and the simplicity of their update algorithms. Let us elaborate on these approaches and the corresponding tree structures that have been developed.

## 4.1   Extending a Balanced Tree Structure

In this approach, Mehlhorn (1979) extended a balanced one-dimensional tree structure to handle weighted one-dimensional data. The basic idea is that a data item with a large weight is represented by a number of leaf nodes so as to maintain the balancing constraint. The only balanced tree structure that has been successfully extended using this approach is a weight-balanced tree (see Nievergelt and Reingold (1973)), resulting in D-trees (Mehlhorn (1979)). D-trees support access, insertion, deletion, promotion, and demotion in logarithmic time per operation in the worst case. They, however, require non-linear storage or else require the use of complicated restructuring operations. On the positive side, they inherit rather strong bounds on their update cost from weight-balanced trees, which makes them useful as underlying data structures for dynamic interval and segment trees (Mehlhorn (1984b)).

## 4.2   Generalizing a Balancing Concept

This approach involves generalizing balanced (one-dimensional) trees to the multidimensional or the weighted-item case. Major balanced tree structures such as B-trees (see Bayer and McCreight (1972)), weight-balanced trees (see Nievergelt and Reingold (1973)) and AVL-trees (Adel'son-Velskij and Landis (1962)) have been generalized. However, because of the various ways a balancing concept can be generalized, a variety of tree structures have been proposed and investigated (see Bent (1982), Bent, Sleator and Tarjan (1980, 1985), Feigenbaum and Tarjan (1983), Gueting and Kriegel (1980), Kriegel and Vaishnavi (1981a), Vaishnavi (1984, 1987)). Fig. 3 summarizes these tree structures. The algorithms for $k$B-trees (Gueting and Kriegel (1980), Gueting and Kriegel (1981), Kriegel and Vaishnavi (1981b) and $kB^+$-trees (Kriegel and Vaishnavi (1981a)) involves a multiplicity of cases and such complex restructuring operations as "Balancing with an indirect brother" and "Collapsing with an indirect brother" (Gueting and Kriegel (1980)). On the other hand, an important attribute of multidimensional AVL trees (Vaishnavi (1983b, 1984)), is that it inherits a number of important properties of the base data structure, i.e., AVL-trees. Its update algorithms involve almost the same restructuring operations as those used in AVL-trees. It has relatively simple algorithms for insertion and deletion. Like AVL-trees, it has the property that insertion of a data item requires the application of at most one restructuring operation. The

| Tree structures | Balancing concepts of B-trees | Balancing concepts of weight-balanced trees | Balancing concepts of AVL-trees | Direct algorithms for insert/delete | Direct algorithms for join/split |
|---|---|---|---|---|---|
| kB-trees | X | | | yes; logarithmic worst-case but complex | |
| $kB^+$-trees | X | | | same as above | |
| Globally biased 2, $b$ trees | X | | | | yes; logarithmic worst-case |
| Globally biased binary trees | X | | | | same as above |
| Globally biased $a$, $b$ trees | X | | | | same as above |
| Biased 2-3 trees | X | | | | yes; logarithmic amortized |
| Biased 2, $b$ trees | X | | | | same as above |
| Biased binary trees | X | | | | same as above |
| Locally biased $a$, $b$ trees | X | | | | same as above |
| Biased weight-balanced trees | | X | | | same as above |
| Pseudo-weight-balanced trees | | X | | | same as above |
| Multidimensional AVL-trees | | | X | yes, logarithmic worst-case | |
| Weighted leaf AVL-trees | | | X | same as above | |

FIG. 3. Summary of tree structures that generalize balancing concepts and require logarithmic worst-case access time.

update simplicity of this tree structure is not at any significant expense of its access time (Vaishnavi (1986)). Weighted leaf AVL-trees (Vaishnavi (1987)), too are characterized by rather simple update algorithms that mirror corresponding AVL-tree algorithms.

## 5.   Paradigms for Modeling Balanced Multidimensional and Weighted Trees

In this section, we describe a paradigm that models the available balanced and semi-balanced multidimensional and weighted structures. It is hoped that the paradigm will guide future research in this area.

The paradigm consists of a strategy used for generalizing a balanced tree structure to higher dimensions, along with a set of strategies for supporting "structure violations" in the generalized structure. We will now describe and illustrate the paradigm.

### 5.1   The Generalization Strategy Paradigm

In the generalized structure, the same balancing concept is used as in the base structure except that "structure violations" that are "supported" (see Vaishnavi (1984)) by certain specified EQSON-subtree(s) are allowed. The last part of the strategy, i.e. the meaning of "support" and the specification of EQSON-subtree(s) that must support a structure violation, is made concrete in such a way that the worst-case height of the generalized structure is logarithmic and logarithmic update algorithms can be designed.

### 5.2   Supporting Structure Violation Strategy Paradigm

We have identified three distinct strategies that may be used for supporting structure violations and that model (along with the general paradigm) the available structure. These strategies may be called the "local-support strategy" (see, for example, Bent (1982), Bent et al. (1980, 1985), the "global-support strategy" (see Bent et al. (1985), Feigenbaum and Tarjan (1983), Gueting and Kriegel (1981)), and the "neighbor-support strategy" (see, for example, Vaishnavi (1984, 1987). The local support strategy leads to update algorithms that are logarithmic in amortized time per operation. The other two strategies lead to update algorithms that are logarithmic in worst-case time per operation.

In each of the above three strategies, a "structure violation" must be "supported" by certain "next-lower-dimensional subtree(s)". The strategies differ in specifying the subtree(s) which must "support" the structure violation. In the local-support strategy, the structure violation must be "supported" by

*each* of the (next-lower-dimensional) *sibling* subtrees of the nodes causing the structure violation. In the *global-support strategy*, the structure violation must be "supported" by *each* of the *adjoining* subtrees of the node(s) causing the structure violation. In the *neighbor-support strategy*, the structure violation must be "supported" by *at least one of the "adjoining"* subtrees of the nodes(s) causing the structure violation.

We shall now briefly describe the structure of $k$B-trees (see Gueting and Kriegel (1980)), and $k$-dimensional AVL-trees (Vaishnavi (1984)), in order to illustrate the generalization strategy described above along with the global-support and neighbor-support strategies, respectively, for supporting structure violations.

## 5.3   Global-Support Strategy Paradigm for $k$B-Trees

$k$B-trees of order $m$, $m \geq 1$, are higher-order generalizations of B-trees of order $m$. Thus 1B-trees of order $m$ are the same as B-trees of order $m$. In a B-tree of order $m$, every node except the root contains $s$ keys, $d \leq s \leq 2d$. In a $k$B-tree, however, there may be nodes which have fewer than $d$ keys (constituting *structure violations*) provided that the structure violations are "supported" by *each* of the "next-lower-dimensional EQSON-subtree which adjoin" the nodes causing the structure violations. A structure violation caused by a node, $N$, at height $h$ is *supported* by a subtree if the height of the subtree is $h$ or more.

Fig. 4(a) shows a 2B-tree of order 1, i.e., a 2-dimensional 2-3 tree. The tree stores the data items:

(a,1), (a,3), (a,7), (a,9), (b,2), (b,4), (b,5), (c,1), (c,2), (d,1), and (d,3).

As in a 2-3 tree, a node, $N$, storing $r$ keys has $r + 1$ subtrees, all of which store keys of the same dimension as those stored in $N$. In addition, each two-dimensional key in a node has a one-dimensional EQSON-subtree. Thus (in Fig. 4a) the subtrees rooted at B and C are EQSON-subtrees of keys $a$ and $b$, respectively. Observe that the left and middle subtrees of the node, A, are empty.

Fig. 4(b) shows the same 2-dimensional 2-3 tree as shown in Fig. 4a except that each empty left, middle, or right subtree of a node $N$ of height $h$ is represented by a chain of $h$-1 unary nodes storing zero keys (shown in the figure as crossed nodes). Observe in Fig. 4b that the subtree rooted at B is an "adjoining subtree of the next-lower dimension" of nodes P, Q, R, and S. Similarly, the subtree rooted at C is an "adjoining subtree of the next-lower dimension" of nodes R, S, D, and T. The structure violation caused by nodes P, Q, R, S, T, U, and V are allowed because each of these structure violations is supported by each of the corresponding adjoining EQSON-subtrees of the next-lower dimension.
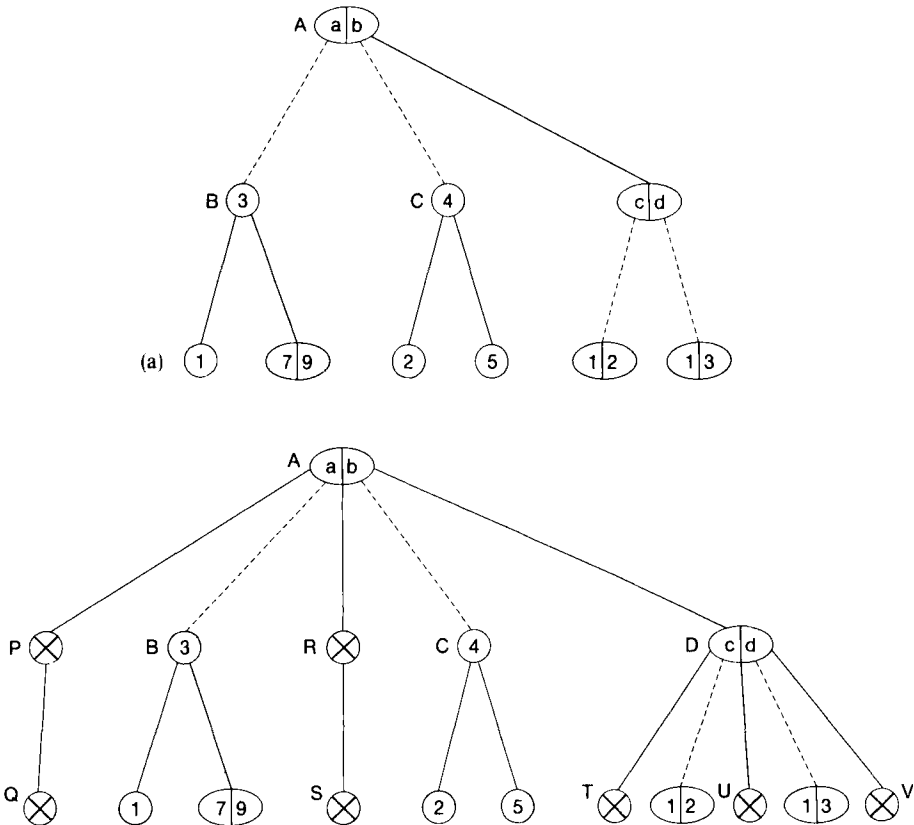
FIG. 4.  (a) shows a 2B-tree of order 1, i.e., a 2-dimensional 2–3 tree; (b) shows the same 2-dimensional 2–3 tree as shown in Figure 4(a) except that each empty left, middle, or right subtree of a node N of height h is represented by a chain of h-1 unary nodes storing zero keys (shown in the figure as crossed nodes).

## 5.4   Neighbor-Support Strategy Paradigm for *k*-dimensional AVL-Trees

*k*-dimensional AVL-trees (kAVL-trees) are higher-order generalizations of AVL-trees. Figure 5a shows a 2AVL-tree. For the sake of explanation, assume that if a node, M, at height $h_1$ has a left or right child at height, N, at height $h_2$, $h_2 < h_i - 1$, then there is a chain of $h_1 - h_2 - 1$ unary nodes, each storing zero keys, between M and N. In particular, assume that each empty left or right subtree of a node at height *h* is represented by a chain of *h*-1 unary nodes. Fig. 5b shows the same tree as in Fig. 5a except for the change in its representation.

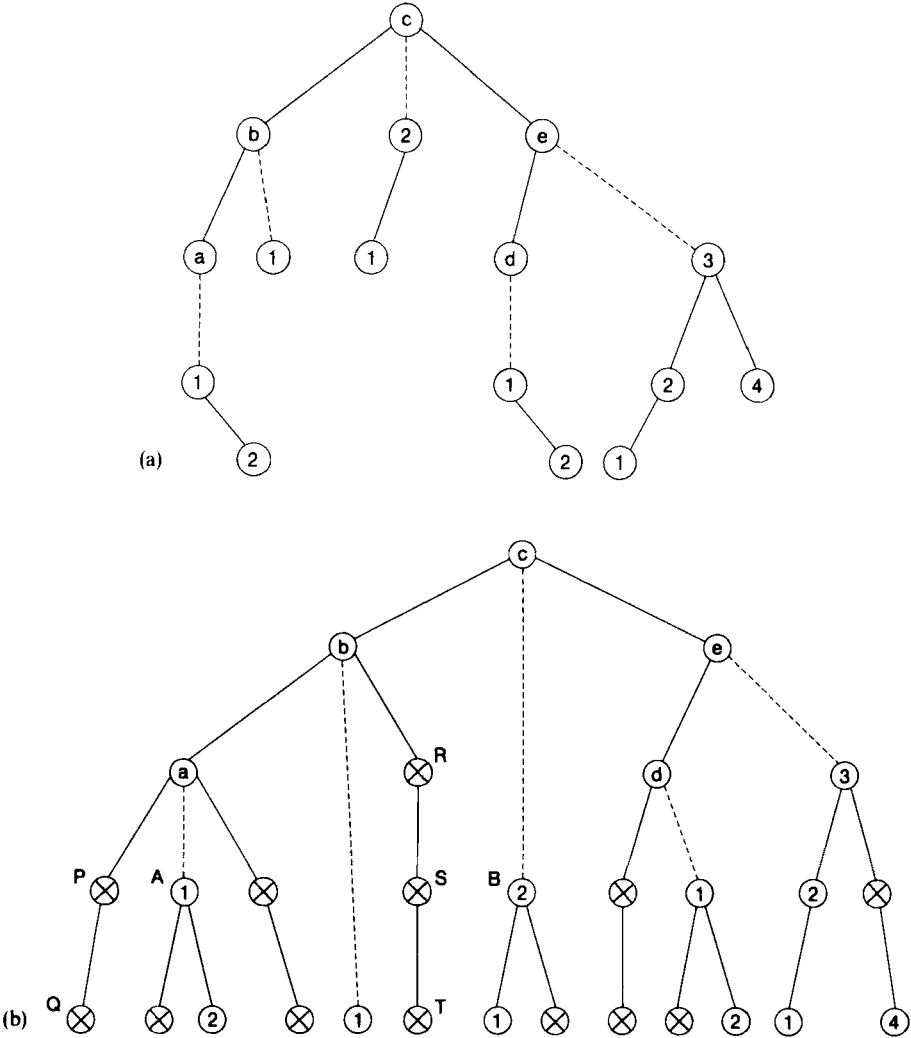FIG. 5.   (a) shows a 2AVL-tree; (b) shows the same tree as in (a) except for the change in its representation.

A chain of *two or more* unary nodes constitute a *structure violation* in a kAVL-tree. A structure violation is allowed in a kAVL-tree provided that it is "supported" by at least one of the "next-lower dimensional EQSON-subtrees that are adjoining" to the nodes in the chain constituting the structure violation.

Let $h$ be the height of the highest node in a chain of unary nodes constituting a structure violation. The structure violation is *supported* by a subtree if the height of the subtree is $h-1$ or more.

In Fig. 5b, the structure violation caused by the chain of nodes P and Q is supported by the EQSON-subtree rooted at A, a one-dimensional EQSON-subtree adjoining to the nodes P and Q. Similarly, the structure violation caused by the chain of nodes R, S, and T is supported by the EQSON-subtree rooted at B. Other structure violations in the tree are similarly supported by appropriate EQSON-subtrees.

Observe that there cannot be a structure violation in a 1AVL-tree, i.e., an AVL-tree, because it is not possible to support it appropriately since the heights of all next-lower-dimensional EQSON-subtrees are zero.

## 6.   Multiple Attribute Trees (MAT)

### 6.1   Preliminaries

The basic abstract data structure underlying the MAT is used in different applications in different forms. An early effort is due to Cardenas and Sagamang (1974) who have extended the concept of the doubly chained tree (DCT) to multidimensional data. Kashyap, Subas, and Yao (1977) and Subas and Kashyap (1975) use a similar notion, but their data structure differs from the abstract data structure in the ranking of the attributes according to the decreasing order of their probabilities of occurrence in a query. There are other structures such as the multidimensional B-trees (MDBT) of Ouksel and Scheuermann (1981), the $k$B trees of Gueting and Kriegal (1980), and the $k$B$^{+}$ tree, and $(k + 1)B^{+}$ trees of Kriegel and Vaishnavi (1981a) and Kriegel (1984)—all these data structures are based on the same abstract data structure. We present a variation of this abstract data structure, which we subsequently call the *Multiple Attribute Tree* (MAT). The data structure, MAT, as used in this paper, is the same as the above mentioned abstract data structure with the additional property that each filial set is ordered according to the attribute values. This is the same as the MAT of Kashyap, Subas and Yao (1977) and Subas and Kashyap (1975), without the attribute ranking constraint. We retain the name because in both structures the most important property, namely the clustering of tree nodes that have "nearby" attribute values, is preserved.

The MAT will be shown to be as efficient as other variants, but is particularly suited for large data structures that are maintained on secondary storage. The details of these structures are deferred to Section 6.2. We formally

define the MAT as follows:

**Definition of MAT.**   A $k$-dimensional MAT on $k$-attributes, $A_1, A_2, \ldots,$ $A_k$ for a set (file) of records is defined as a tree of depth k, with the following properties:

(i) It has a root at level O.

(ii) Each child of the root is a $(k\text{-}1)$-dimensional MAT, on $(k\text{-}1)$ attributes, $A_2, A_3, \ldots, A_K$, for the subset of the records that have the same $A_1$ value. This value of $A_1$ is the value of the root of the corresponding $(k\text{-}1)$-dimensional MAT.

(iii) The child nodes of the root are in the ascending order of their values. This set of child nodes is called the *filial set*.    ■

Fig. 6a shows a set of records, and the corresponding MAT is shown in Fig. 6c. We note that there is a root node at level 0, which does not have any value. Every node of level i, i = 1, 2, ..., $k$ corresponds to a value of attribute $A_i$ and this value is the value of the node. Thus, the attributes $A_1, A_2, \ldots, A_k$ form the hierarchy of levels 1 through $k$. The important property of the MAT is that every filial set is ordered. This is the result of the property that the sorting of a set of records on all the attributes naturally induces a tree structure, as illustrated in Fig. 6b. Hence the abstract MAT can be imagined to have been constructed as follows:

a) Records are sorted in ascending order on all attributes.

b) Starting from the first attribute, all the elements of an attribute having the same attribute value are combined into a node. This process is recursively carried out for each subset of records that have the same first attribute value.

| $A_1$ | $A_2$ | $A_3$ | $A_4$ | Record pointer |
|-------|-------|-------|-------|----------------|
| 1 | 1 | 3 | 3 | 1 |
| 1 | 2 | 1 | 2 | 2 |
| 1 | 1 | 4 | 1 | 3 |
| 1 | 1 | 2 | 6 | 4 |
| 2 | 3 | 5 | 7 | 5 |
| 1 | 1 | 3 | 5 | 6 |
| 1 | 2 | 5 | 6 | 7 |
| 2 | 3 | 5 | 1 | 8 |

(a)

Fig. 6.   (a) Input data file; (b) Sorted input file, and induction of a natural tree structure; (c) MAT representation for data of (a).
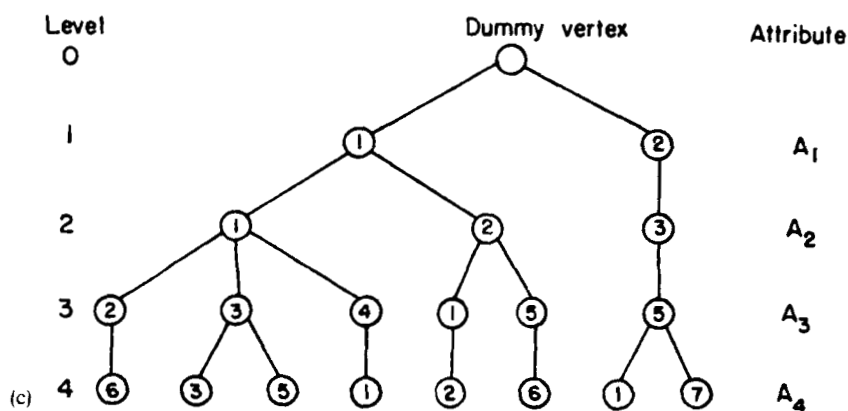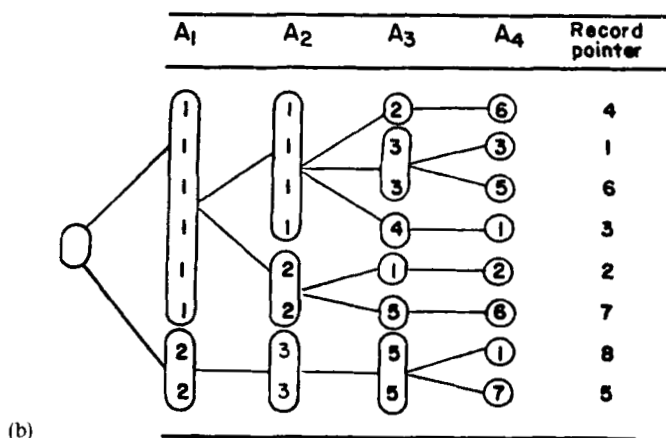
(b)



(c)

FIG. 6.  (*Continued*)

The construction of the MAT can be visualized using geometric notions. The sub-MAT on $A_2, A_3, \ldots, A_k$, generated as in *ii*) of the definition above, corresponds to the set of points that lie on a hyperplane in the data space. The equation of this hyperplane is $A_1 = a_1$ is the value of $A_1$ corresponding to the root of the sub MAT. Since a hyperplane in a space of dimensionality $k$ corresponds to a space of dimensionality $(k - 1)$, each recursive generation of a sub-MAT is carried out on a space of lesser dimensionality. The same effect will be found in the search algorithms, which recursively search on spaces of lesser dimensionality.

Another important property is that each record or point is represented by a *unique* path from the root to the corresponding terminal node. Because of this property the profile of the MAT, for a set of records, completely depends on the occurrence of various attribute values in the records. In general, the sizes of various filial sets depend on the exact set of input records, and the corresponding attribute values. As a consequence of this behavior the sizes of filial-sets are bound to be random. Hence, we characterize the sizes of the filial-sets of the MAT by average measures. This type of analysis is done for the MAT by Rao and Iyengar (1986) and by Ouksel and Scheuermann (1981) for the multidimensional B-trees. These measures may correspond to the expected values of filial-set sizes, if the probability distribution of the various attributes is known. The disparity of the actual filial set size from this average value can be taken into account by using dispersion measures such as variance, third moment, etc.

We use the following notation in our subsequent discussion:

$N$: number of records

$M$: number of nodes of the MAT

$k$: number of attributes

$s_j$: average (expected) size of the filial-set at level $j$,   $j = 1, 2, \ldots, k$

$\sigma_j$: standard deviation (square root of variance) of the filial set at level $j = 1, 2, \ldots, k$

$d_j$: size of the set from which values of attribute $A_j$ can be chosen,   $j = 1, 2, \ldots, k$

The properties of the MAT can be summarized as follows:

$$s_0 = 1$$

$$M \leq (1 + kN)$$

$$N = s_1 s_2 \cdots s_k$$

$$N = \prod_{i=1}^{k} s_i \tag{6.1}$$

$$M = \sum_{j=1}^{k} \prod_{i=1}^{j} s_i \tag{6.2}$$

When the distribution of data is symmetric over attributes, we characterize the MAT as the *symmetric* version, where the average filial set sizes of all levels are the same, i.e., $s_1 = s_2 = \cdots = s_k = s$, $N = s^k$. This characterization is most appropriate when the distribution of each attribute values is independent of

the distribution of other values, and uniform with the same mean. For example, let each $A_i$ correspond to an independent and uniform distribution with probability of occurrence of each value to be $p$. Then for a symmetric MAT we have $s = pd$, and $d_1 = d_2 = \cdots = d_k = d$. For a symmetric MAT we have

$$s_j = O(N^{1/k}) \tag{6.3}$$

$$\prod_{j=1}^{k-1} s_j = N/s_k = O(N^{1-1/k}) \tag{6.4}$$

In the following sections, we use the symmetric MAT for our analysis. Without loss of generality, the expressions (6.3) and (6.4) are used to estimate the complexity of various algorithms. A MAT for any data can be imagined to be equivalent to a symmetric version with the order estimates satisfying (6.3) and (6.4). For many practical situations, this approximation gives reasonably good analytical measures, as reported in Veni Madhavan (1984), and Rao and Iyengar (1986). However, it is to be noted that there could be certain applications in which this approximation is not completely accurate. But this is the approach that is used mostly in the literature as in Kashyap et al. (1977) and Gopalakrishna and Veni Madhavan (1980). A completely generalized analysis seems very difficult at this stage.

## 6.2   MAT-Related Multidimensional Data Structures

There are many variants of the MAT data structure. Most of these data structures differ from one another in the organization of either pointers or filial sets.

The data structure proposed by Cardinas and Sagamang (1974) is a refinement of the doubly chained tree (DCT) of Sussenguth (1963). In the modified DCT each level corresponds to an attribute, and Fig. 7 shows the basic idea of the doubly chained tree for the data of Fig. 6a. In this data structure, more pointers are used to facilitate the search and other operations. One disadvantage of the DCT is that the length of the path corresponding to a single record can at most be $N + k$, as against $k$ for the other data structures such as the MAT, etc. In the worst case, the search path for a complete match query can contain as many as $N + k$ nodes. Kashyap et al (1977) have shown the superiority of the MAT over the modified DCT.

The other structures such as the MDBT, $k$B-tree, etc. are analogous to the MAT in the basic organization of data. But they differ in the organization of filial sets. The organization of filial-sets plays an important role in the performance of the data structure. The binary search tree complex (BST-complex) of Lien et al. (1975) is one of the early structures of this kind. The

**Level**                                                                      **Attribute**
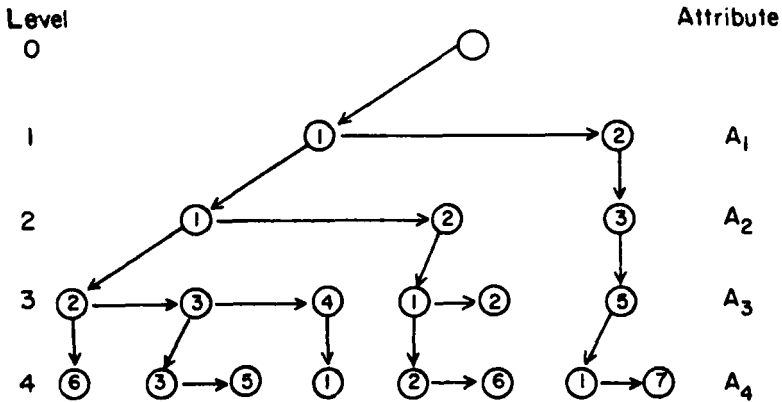


FIG. 7.   Doubly chained tree for data of Fig. 6a.

BST-complex for a file of records is similar to the MAT. In the BST-complex, each filial set is maintained as a height-balanced binary tree or an AVL-tree. (In an AVL-tree the difference between the heights of the left and right subtrees of every node is at most one.) This guarantees logarithmic searching time for a complete match query. Fig. 8b shows the BST complex for the datafile of Fig. 8a. The leaves carry pointers to the page numbers. Thus, if the number of distinct key values at level $i$ is $v(i)$, the time required for the BST-complex to respond to an exact match query is at most $\sum_{i=0}^{k-1} \log(v(i)) \le k \log N$. Despite its good performance for complete match, the BST-complex does not efficiently support partial match and range queries.

The multidimensional B-tree (MDBT) of Ouskal and Scheuermann (1981) ensures efficient searching for various associative queries for database environments. The central idea is the use of B-trees to maintain the filial sets in a structure similar to MAT. This structure, called the MDBT, is proposed as a viable alternative for an environment with a fairly high degree of query complexity. Strategies to maintain the MDBT are also discussed in Ouksel and Scheuermann (1981). Analytical estimates for random MDBT are given by $O(\log N)$, $O(N \log N'/N')$, and $O(\log N)$ as complexities for the worst-case exact match, worst-case partial match and restricted average range query respectively. $N'$ denotes the product of average filial set sizes of unspecified levels for a partial match. It is to be noted that the MDBT is assumed to be in an 'average' form. Again, the restricted range query is a simplified form of a generalized range query. Each attribute is assumed to specify no more than two attribute values. However, a generalized range query will have substantially higher complexity in the MDBT than $O(\log N)$.

Another tree based index structure which is based on the same concepts as the MDBT is the multidimensional B-tree ($k$B-tree) proposed by Gueting and

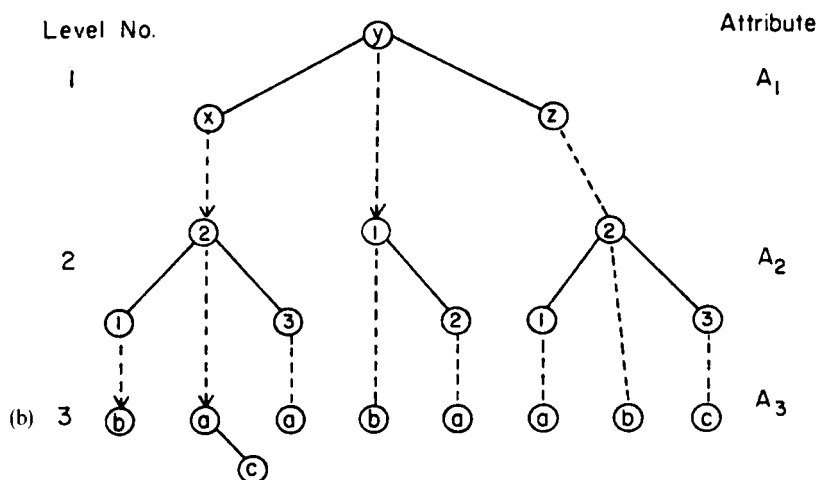| $A_1$ | $A_2$ | $A_3$ | Page No. |
|-------|-------|-------|----------|
| x | 1 | b | 1 |
| y | 1 | b | 2 |
| x | 2 | a | 3 |
| y | 2 | a | 4 |
| x | 2 | c | 5 |
| x | 3 | a | 6 |
| z | 2 | b | 7 |
| z | 1 | a | 8 |
| z | 3 | c | 9 |

(a)



(b)

FIG. 8.   (a) Input data file; (b) BST-complex for data in (a).

Kriegel (1980, 1981). Under the assumption of a uniform and independent distribution of the attributes the MDBT will have a maximal height of log $N + k$. But on the data that does not satisfy these properties, the search depth could be as much as $O(k \log N)$. The novel method of "balancing over all components" in the $k$B-tree ensures the depth to be $O(\log_{m+1} N + k)$, where $m$ is the order of the $k$B-tree. Hence, the complete match query, insertion and deletions restricted to one path have logarithmic time complexity. We will present an example from Kriegel (1984) to illustrate the difference between the MDBT and $k$B-tree. Figure 9a shows two-dimensional records as points in a two-dimensional space. Figure 9b shows the corresponding MDBT. In the
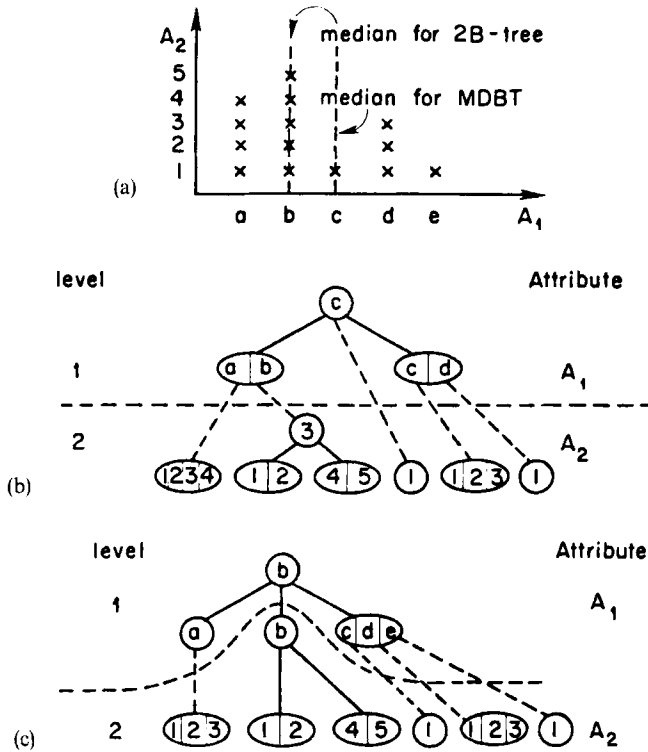
FIG. 9.   (a) Input data points; (b) MDBT for data of (a); (c) *k*B-tree for data of (a).

MDBT each level corresponds to an attribute, and each filial-set is maintained as a B-tree at the same level. On level 1 the median of *a*, *b*, *c*, *d*, *e*, is *c*, as balancing is based on attribute 1 alone. Figure 9c shows the *k*B-tree. At level 1, *b* is selected as the median, as balancing is based on all the records. This notion of balance is shown to ensure the $O(\log N + k)$ depth for the *k*B-tree. Though the concept of balancing over all components can be found in Lee and Wong (1977), the main contribution of the *k*B-trees is their maintainability in dynamic environments. However, both the MDBT and the *k*B-tree have a shortcoming: neither supports sequential processing at each level. This property is very important for answering partial match and range queries. There are other refinements of these on the *k*B-tree such as the $kB^+$-tree, and $(k + 1)B^+$-tree, which are more efficient in certain ways. See Kriegel (1984) for details.

We note that the main difference between the data structures discussed above lies in the way in which the filial-sets are organized. In the MAT, the filial-sets are maintained as ordered lists which support binary search. In main

memory this feature is naturally supported if the members of filial-sets are stored in consecutive locations. Consequently, searching a filial-set of size $s$, for a member costs almost $O(\log(s))$. The exact implications of this aspect are discussed in the next section.

## 6.3   Directory Generation

The data structure MAT is linearized to form a directory. The directory defines the way various levels are organized and the way the filial sets are maintained. The data structures such as the MAT, DCT, MDBT, $k$B-tree, $k$B$^+$-tree, etc. can be implemented using pointers. Answering a query calls for the traversal of pointers. In general, no assumption is made on the way the pointers are organized. This type of implementation using pointers is efficient when the data structure resides in the main memory. But when the directory resides in secondary memory, where the transfer of information is in the form of pages, the randomness of the pointer organization results in the retrieval of many unnecessary pages. For example, let us consider the case in which the nodes of a filial set reside on separate pages in secondary memory. This situation might arise when the filial set is generated by dynamic insertions of records after the static structure is constructed. In this case, the searching for a value in a filial set of size $s$ is likely to result in $O(\log N)$ retrievals of secondary memory pages. But instead, if a filial set is maintained on a single page or on a constant number of pages, then the search involves $O(1)$ retrievals of secondary pages. Once a secondary page is loaded into main memory, search can be carried out on the filial set for the required value. This situation arises in physical database organizations where the files physically reside on secondary storage. The other applications may include graphics, computational geometry, etc., where the large sizes of data necessitate storage of information on secondary memory.

The main driving force for the MAT-based organization is secondary memory storage. Though the abstract structure MAT can be implemented using pointers, we consider an organization where the nodes corresponding to a filial set are kept together. The main idea is to cluster the nodes that are likely to be accessed in answering a query. In a complete match query, the nodes of a filial set are searched for a value. In a partial match query, either the filial set is searched for a value, or the whole of the filial set is retrieved. In a range query, the nodes within the specified range are retrieved. In order to answer the above queries, we introduce the concept of linearization, which enables us to store the MAT data structure in the form of a table. There are two basic types of linearization as proposed by Subas and Kashyap (1975) and Kashyap et al. (1977): depth-first linearization and breadth-first linearization. Either can be done in a top-down or bottom-up manner. The top-down method is more

efficient than the bottom-up method. This is because, in answering a query, the top-down method results in reducing the number of nodes to be searched. In the depth-first organization, the nodes corresponding to the fields of a record are kept together, as illustrated in Fig. 10a. The numbers by the side of the node represent the entry number of the nodes in the directory. This is an efficient technique for full match queries, as in the best case the answer can be produced in a single access to secondary storage (assuming $k$ nodes can be stored in a single page). However, in the worst case, $k$ accesses are needed. But this does not naturally support the partial match and range queries, as members of a filial set are not kept together. In the breadth-first linearization, the nodes corresponding to a filial set are kept together as shown in Fig. 10b. This method naturally supports the partial match and range queries.
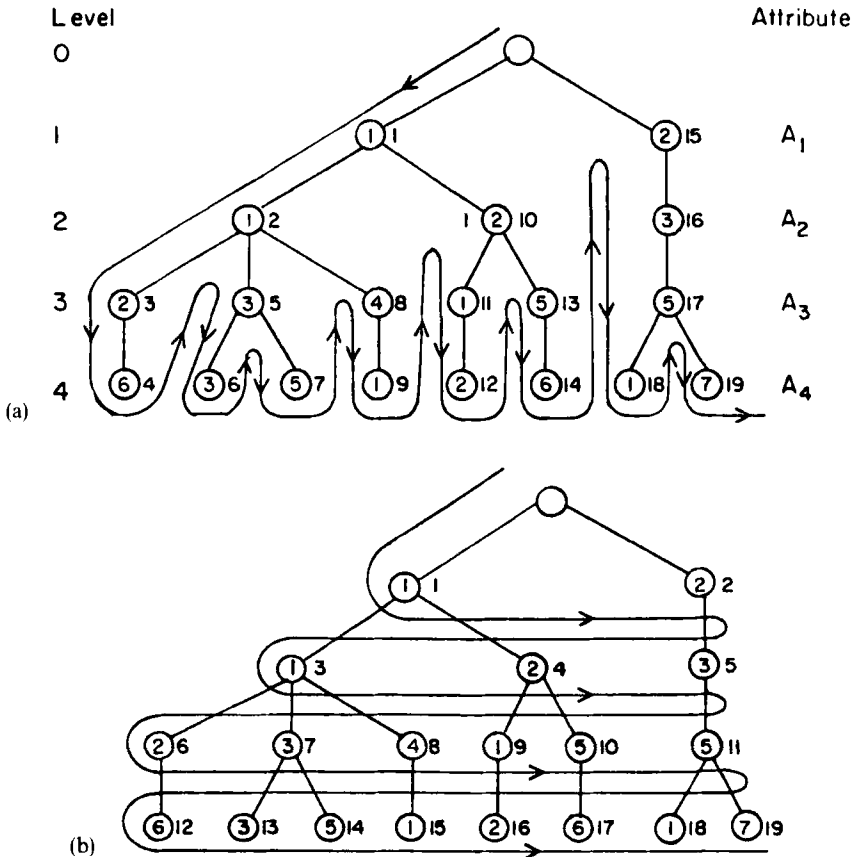


FIG. 10.   (a) Depth-first linearization; (b) breadth first linearization.

Figures 11a and 11b show the directories corresponding to the depth-first and breadth-first linearizations, where each directory element is of the following form:

$$directory\ element = \textbf{record}$$

$$\text{node-num: } 1..M;$$

$$\text{value: } 1..M;$$

$$\text{first-child/brother: } 1..M;$$

$$\text{last-child/cousin: } 1..M;$$

$$\textbf{end};$$

where the fields corresponding to a node $T$, at level $j$, and numbered $n$ are defined as follows:

value: value of node $T$;

first-child: node number of the first element
of the child set of node $T$;     for $j = 1, 2, \ldots, k$

record pointer           for $j = k$;

last-child: node number of the last element
of the child set of node $T$;     for $j = 1, 2, \ldots, k$

: 0                   for $j = k$;

brother: node number of the node next to
node $T$ in the order;      for $j = 1, 2, \ldots, k$;

cousin: node number of the first node of
the filial set next to that of
node $T$;            for $j = 1, 2, \ldots, k$;

The fields *first-child* and *last-child* are used in the breadth-first linearization. The fields *brother* and *cousin* are used in the depth-first linearization. The algorithms to generate a depth-first directory can be obtained from Kashyap *et al.* (1977), and Gopalakrishana and Veni Madhavan (1980). We give an algorithm to generate a breadth-first directory.

Step 2 of DIR-GENERATION computes base$[i]$, $i = 1, 2, \ldots, (k+1)$, such that any directory node at level $i$ lies between base$[i]$ and base$[i + 1] - 1$. These values are needed to store the value, first-child and last-child fields in algorithm DIR-GENERATION. This algorithm is given below.

The main cost of algorithm DIR-GENERATION is due to sorting. After the imput records and sorted, two passes are made on the sorted records, once

| Node-number | Value | Brother | Cousin |
|:---:|:---:|:---:|:---:|
| 1 | 1 | 15 | − |
| 2 | 1 | 10 | 16 |
| 3 | 2 | 5 | 11 |
| 4 | 6 | − | 6 |
| 5 | 3 | 8 | 11 |
| 6 | 3 | 7 | 9 |
| 7 | 5 | − | 9 |
| 8 | 4 | − | 11 |
| 9 | 1 | − | 12 |
| 10 | 2 | − | 16 |
| 11 | 1 | 13 | 17 |
| 12 | 2 | − | 14 |
| 13 | 5 | − | 17 |
| 14 | 6 | − | 18 |
| 15 | 2 | − | − |
| 16 | 3 | − | − |
| 17 | 5 | − | − |
| 18 | 1 | 19 | − |
| 19 | 7 | − | − |

(a)

FIG. 11.   (a) Depth-first directory; (b) breadth-first directory.

for algorithm COMPUTE-BASE and once for lines 3 to 11 of algorithm DIR-GENERATION. If the input resides on the main memory, then the sorting phase has $O(kN \log(kN))$ comparisons. Algorithm COMPUTE-BASE takes $O(kN)$ comparisons, or memory accesses. This is because line 6 of COMPUTE-BASE and line 6 of COMPUTE-BASE and line 6 of DIR-GENERATION each involve at most $k$ comparisons for each record. Again, line 8 of COMPUTE-BASE is executed at most $k$ times for each record. Similarly, we can show that line 8 and line 9 of DIR-GENERATION each can be executed at most $k$ times for each record. Hence the complexity of

| Node-number | Value | First-child | Last-child |
|---|---|---|---|
| 1 | 1 | 3 | 4 |
| 2 | 2 | 5 | 5 |
| 3 | 1 | 6 | 8 |
| 4 | 2 | 9 | 10 |
| 5 | 3 | 11 | 11 |
| 6 | 2 | 12 | 12 |
| 7 | 3 | 13 | 14 |
| 8 | 4 | 15 | 15 |
| 9 | 1 | 16 | 16 |
| 10 | 5 | 17 | 17 |
| 11 | 5 | 18 | 19 |
| 12 | 6 | 4 | — |
| 13 | 3 | 1 | — |
| 14 | 5 | 6 | — |
| 15 | 1 | 2 | — |
| 16 | 2 | 2 | — |
| 17 | 6 | 7 | — |
| 18 | 1 | 8 | — |
| 19 | 7 | 5 | — |

(b)

FIG. 11.  (*Continued*)

DIR-GENERATION  is  $O((kN) \log(kN) + 2kN) = O((kN) \log(kN))$.  If each record comparison is taken to be of unit time then the complexity is $O(N \log N)$.

If the input records reside on secondary storage, then the cost is due to the page accesses during sorting. The other parts of the algorithm make two sequential passes over the sorted data, and $O(m)$ page accesses are made during these passes, where $m$ is the number of pages in which data is stored. Hence, using standard external sorting techniques, the directory generation can be done in $O(m \log m)$ page accesses as in Horowitz and Sahni (1976).

---

**algorithm** DIR-GENERATION;
**begin**
1.    sort the input data file on all $k$ attributes
2.    COMPUTE-BASE;
3.    read first record;
4.    **repeat**
5.      read next record;
6.      compare it with previous record to get $L$,
        the highest level at which the attribute
        values are different;
7.    **for** $j := L$ **to** $k$ **do**
8.      assign value field;
9.    **for** $j := (L + 1)$ **to** $k$ **do**
10.    assign first-child and last-child fields;
11.    **until** all records are processed;
**end**;

---

**algorithm** COMPUTE-BASE;
**begin**
1.    **for** $i := 1$ **to** $(k + 1)$ **do**
2.      $base[i] := i$;
3.    read first record;
4.    **repeat**
5.      read next record;
6.      compare it with previous record to get $L$,
        the highest level at which the attribute
        values are different;
7.    **for** $j := L$ **to** $(k + 1)$ **do**
8.      $base[j] = base[j] + 1$;
9.    **until** all records are processed;
10.    **for** $j := 2$ **to** $(k + 1)$ **do**
11.    $base[j] = base[j] + base[j - 1]$;
**end**;

---

## 6.4   Search Algorithms

In this section, we dicuss MAT-based search algorithms for complete match, partial match and range queries. The concept of "clustering" of data is exploited to improve the average case performance of various search

algorithms. This is achieved by keeping together the nodes that are likely to be accessed in answering a given query. The analysis of the performance is carried in the following two cases:

(i) directory resides on the main memory,
(ii) directory resides on the secondary storage.

The breadth-first linearization assures that the members of a filial set are consecutive in main memory and a binary search can be carried out on them. When the directory resides on secondary memory, this clustering helps in reducing the number of pages accessed. We assume that each filial set occupies no more than a constant number of pages on secondary memory. We also assume that a filial set can be completely loaded into the main memory when queries are being answered. We show in this section that the MAT fares almost as well as other structures, such as the $k$B-tree, MDBT, etc., when the directory resides on main memory. Our main result is showing that the MAT is a better choice when the directory resides on secondary memory.

We define a query $Q$, as follows:

$$Q = \bigcap_{i=1}^{k} q_i$$

where $q_i$ is the qualifier for atrribute $A_i$. For a complete match query, $q_i$ specifies a value for $A_i$, $i = 1, 2, \ldots, k$. For a range query, $q_i$ specifies a range $[l_i, h_i]$ for attribute $A_i$, $i = 1, 2, \ldots, k$, where $l_i$ and $h_i$ specify the lower and upper limits of the range respectively. For a partial match, each level $i$. $i = 1, 2, \ldots, k$, could be specified or unspecified. For a specified level $i$, $q_i$ specifies a value for $A_i$. For unspecified levels, $q_i$ does not specify any restriction on $A_i$. A full match can be visualized as a range query that specifies the range of a single value for each attribute. Similarly, a partial match query can be visualized as a range query that specifies a range of size one for specified levels, and range of the domain of the attribute for unspecified levels. Normally, a complete match and partial match are easier to answer than a range query. We also discuss a restricted range query, which specifies no more than $a$ values for each attribute. Answering a query can be visualized using geometric notions: the set of records of $k$ attributes describes a $k$-dimensional space. A range query describes a rectilinearly oriented hyperrectangle in $k$ dimensions. Answering a range query corresponds to the retrieval of the points contained in the hyperrectangle specified by the range query. In the MAT-based organization, while answering a complete match, partial match or range query, the processing of each attribute reduces the dimensionality of the space to be searched by one.

The basic notion of a MAT-based search is that of "hierarchical descent", where the query is processed level by level starting from the root node. At each level $j, j = 1, 2, \ldots, k$, nodes that satisfy the 'partial' query $\bigcap_{i=1}^{j} q_i$ are aggregated. These nodes are called *qualified nodes*. The qualified nodes at level $k$ contain pointers to the physical records that satisfy the query. We restrict our discussion to the breadth-first linearization. More details on this can be found from Rao *et al.* (1984). The discussion on depth-first linearization can be found in Gopalakrishana and Veni Madhavan(1980).

In the algorithm MAT-SEARCH, line 10 corresponds to the initial call. When the directory resides on secondary storage, the execution of lines 2 and 3 is very critical in deciding the number of page accesses. The set qnodes of MAT-SEARCH is sorted and accessed in order. This results in accessing the consecutive filial sets from left to right in the MAT structure. This type of access is efficient when the sets are of smaller sizes such that a number of them can be accommodated in a single page. This scenario occurs in answering a range query. In fact, using breadth-first ordering and a sorted qnodes set, sequential accesses of the directory can also be supported efficiently.

---

**algorithm** MAT-SEARCH(level, qnodes);
**begin**
1.    tempset := $\phi$;
2.    **for** each $n \in$ *qnodes* **do**
3.        add to tempset all children of $n$ that
           satisfy $q_{level}$;
4.    **if** level $< k$
5.    **then** MAT-SEARCH(level $+ 1$, tempset)
6.    **else**
       **begin**
7.        **for** each $n \in$ tempset **do**
8.        retrieve record pointer;
9.    **end**;
**end**;
10.   MAT-SEARCH(1, {root});

---

*a) Complete Match Query.* For a complete match query, at any level $j$, there is only one node that satisfies the partial query $\bigcap_{i=1}^{j} q_i$ since $\bigcap_{i=1}^{j} q_i$ specifies a unique combination of attribute values in the 'partial' MAT curtailed to the first $j$ levels. Hence line 3 of algorithm MAT-SEARCH is executed only once for each level. For each execution of line 3, a binary search is carried out on the filial set for the specified attribute value. Hence the worst-

case complexity of the complete match query is given by

$$O(\log(v_1) + \log(v_2) + \cdots + \log(v_k)) = O(k \log N).$$

If there is "uniformity" in the profile of MAT, such that the average size of the filial set of level $j$ is $s_j$, then the average complete-match query cost becomes

$$O(\log(s_1) + \log(s_2) + \cdots + \log(s_k)) = O(\log N).$$

This estimate is appropriate for random data satisfying certain distributions. The complexity of $O(\log N)$ for a complete match for multidimensional data can be easily achieved by treating it as an equivalent single-dimensional problem. The key corresponding to any record in the equivalent problem is obtained by concatenating the corresponding attribute values. A balanced binary tree can be easily constructed to ensure $O(\log N)$ complexity for searching any key corresponding to the complete match query. However, such a strategy is not suited for handling partial match and range queries.

If the directory is maintained in secondary storage, at each level we retrieve no more than $c$ pages corresponding to the filial set of the qualified node at the previous level. Hence the number of page accesses for a complete match is $O(ck) = O(k)$. In breadth-first organization $ck$ pages are always retrieved for any complete match query. For depth-first linearization, in the best case, one page access will be sufficient to answer a complete match query. In this case all the qualified nodes corresponding to a complete match query lie on a single page. But in the worst case $O(s_1 + s_2 + \cdots + s_k)$ pages may be retrieved. In this case, the qualified nodes are the last ones in their filial sets. Using the average characterizations of the MAT, the worst-case number of page accesses is $O(kN^{1/k})$ for a complete match query. It is interesting to note that this many page accesses will be needed when the MAT is implemented using pointers and no clustering is assumed on the nodes.

*b) Partial Match Query.* The maximum number of records that satisfy a partial match query that specifies $t$ levels is given by

$$t_1 t_2 \ldots t_k$$

where

$$t_i = \begin{cases} 1 & \text{if } A_i \text{ is specified} \\ v_i & \text{if } A_i \text{ is not specified} \end{cases}$$

Using the average characterizations of Section 6.1, we can conclude that the number of records that satisfy a given partial match query is $O(N^{1 - 1/k})$ for uniformly distributed data. In line 3 of algorithm MAT-SEARCH, for

specified levels, we carry out binary search on the filial sets. For unspecified levels, we qualify the entire filial set. In the worst-case situation for our algorithm the first $(k - t)$ attributes are unspecified, and each qualified node at level $(k - t)$ gives rise to one qualified node at each subsequent level. Hence for each level $j$, $(k - t + 1) < j < k$, we have $O(N^{1 - t/k})$ sets to be searched, and at each level there can be $O(N^{1 - t/k})$ qualified nodes. Hence the cost of answering a partial match query is given by

$$O(s_1 s_2 \cdots s_{k-1} (\log(s_{k-t+1}) + \cdots + \log(s_k))) = O(t N^{1 - t/k} \log(N^{t/k}))$$

We note that the above expression approaches $O(\log N)$ as more and more attributes are specified. But, as we leave some of the attributes unspecified, the complexity increases. This is because more and more nodes could get qualified for a partial match as fewer and fewer attributes are specified.

If the directory is maintained on secondary storage the number of page accesses corresponding to the first $(k - t)$ unspecified levels is $O(N^{1 - (t - 1)/k}$— one page access for each accessed filial set. The number of pages accessed in the subsequent $t$ specified levels is $O(t N^{1 - (t - 1)/k})$. Hence the total number of page accesses is $O(t N^{1 - (t - 1)/k})$. As more and more attributes are specified, this cost approaches $O(k N^{1/k})$. If no linearization is assumed, each qualified node might give rise to a page access, and thus the number of page accesses is $O(t N^{1 - t/k} \log(N^{t/k}))$. If depth-first ordering is used for linearization, the number of page accesses could be shown to be $O(t N^{1 - t/k})$. The performance of breadth-first linearization is marginally better than that of depth-first linearization. However, both linearizations are more efficient than the random implementation. It is to be noted that there exists a certain amount of clustering of nodes on the secondary pages, and the assumption of complete randomness on the nodes allocation is not valid. However, it provides a theoretical upper bound on the number of page accesses.


*c) Range Query.* The performance analysis of a range query based on the worst case is mainly of theoretical significance rather than of great practical value. Specifically, in the worst case, a range query may retrieve all the $N$ records, which is not very realistic, especially in large database environments. Normally, the number of records retrieved is small, and the queries that retrieve a large number of records are infrequent. Hence an approximate estimate for the number of retrieved records that is close to the average value is more appropriate to describe the performance of the system. However, the average case analysis is difficult because the exact average case patterns of a query are not known and also the average case varies from problem to problem. In this section we first consider generalized range queries and then discuss restricted range queries.

Normally, the time complexity of a range query is expressed in the form

$$O(f(N) + K_1),$$

where $k_1$ is the number of records retrieved and $f(N)$ gives the cost involved in searching for the limits that delimit the qualified record pointers. The main efforts in designing efficient algorithms for range search are directed towards decreasing $f(N)$ without considerable increase in storage and preprocessing costs. For a range query, step 3 of the algorithm MAT-SEARCH is modified to search for the upper and lower limit of the specified range for that level. The nodes that lie within the limits are retrieved and added to tempset. This feature is well supported by the breadth-first linearization.

In the worst cast of the range query, all the nodes of the searched fiial sets are qualified. Hence for a filial set of size $s$, the cost of searching for limits is $O(\log(s))$ and the retrieval of the qualifid node costs $O(s)$. Using the average characterization of the MAT, the search cost at level $j$ can be computed as $((\prod_{i=1}^{j-1} s_i) \log(s_j))$. Since, all the nodes of a level are qualified in the worst case, the cost of processing a range query for level $j$ is $O((\prod_{i=1}^{j-1} s_i) \log(s_j) + \prod_{i=1}^{j} s_i)$. Hence, the time complexity of a range query is given by $\sum_{j=1}^{k-1} (\prod_{i=1}^{j} s_i) + (\prod_{i=1}^{k-1}(s_i) \log(s_k) + K_1$ where the first term corresponds to the cost of processing the first $(k-1)$ levels. The second term corresponds to the cost of searching for the range limits for the final level. The third term, namely $K_1$, corresponds to the number of nodes qualified in the final level. The cost of a range query can be simplified and given as follows:

$$O(N^{1-1/k} \log(N^{1/k}) + K_1).$$

If the directory resides on secondary storage, arguments similar to those above can be used to show that the number of page accesses is $O(N^{1-1/k})$. If no linearization is used, and random location of nodes among secondary storage pages is assumed, then the number of page accesses is $O(N^{1-1/k} + K_1)$. However, as remarked earlier, such an assumption is not very practical. It is again interesting to note that depth-first ordering also has a range search complexity of $O(N^{1-1/k} + K_1)$ page accesses.

So far we have considered only the worst-case estimates for a range query. The assumption that all members of filial sets satisfy the query is not realistic. If we impose a restriction that for a range query no more than $a$ nodes get qualified in each filial set searched, then the complexity of a range query can be given as:

$$\sum_{j=1}^{k-1} \prod_{i=1}^{j} a^i + \left(\prod_{j=1}^{k-1} a^j\right) \log(s_k) + K_1$$

which can be approximated (on the average) by

$$O(a^{k-1} \log(N^{1/k}) + K_1).$$

Let us define $p$, called the *qualified fraction*, to be

$$p = \frac{a + a^2 + \cdots + a^k}{N}$$

$$= \frac{a(a^k - 1)}{(a - 1)N}$$

$$= \frac{a^k}{N}, \quad \text{for reasonably large } a.$$

The qualified fraction gives an average estimate of the maximum value of the fraction of the nodes that get qualified from a filial set. Now the cost of the average range query is given by

$$O((p(N)^{(k-1)/k} \log(N^{1/k}) + K_1) \approx O(pN^{1-1/k} \log(N^{1/k}) + K_1)$$

Hence, the value of $p$ determines the cost of the range query. When the directory resides on secondary storage, the number of page accesses is given by $O(pN^{1-1/k})$.

The intuitive feel for this average profile of a range query given by Rao and Iyengar (1986) is that, on the average, a range query defines a "sub-MAT" that has a filial-set size of $ps_i$ at level $i, i = 1, 2, \ldots, k$. This visualization gives more insight into the process of answering a range query on MAT. Table I summarizes the order of the cost of processing a range query for various values of $p$.

The performance of the MAT becomes better as the number of records retrieved by a range query is reduced. This is generally the case for applications

TABLE I

AVERAGE COST ANALYSIS FOR THE RANGE QUERY OF TYPE $Q = \bigcap_{i=1}^{k} q_i$

($K_1$ denotes the number of qualified records)

| Qualified fraction $p$ | Query cost main memory | Query cost secondary memory |
|---|---|---|
| $\left(\dfrac{N}{N}\right)$ | $O(N^{1-1/k} \log(N^{1/k}) + K_1)$ | $O(N^{1-1/k})$ |
| $\left(\dfrac{\log N}{N}\right)$ | $O(\log N + K_1)$ | $O(\log N)$ |
| $\left(\dfrac{\log \log N}{N}\right)$ | $O(\log(N^{1/k} + \log N) + K_1)$ | $O(\log(N^{1/k} + \log N))$ |
| $\left(\dfrac{c}{N}\right), c = \text{constant}$ | $O(\log(N^{1/k}) + K_1)$ | $O(\log(N^{1/k}))$ |

involving large amounts of data, such as large databases, image processing systems, etc.

We conclude this section by noting that when secondary storage is used for the directory the appropriateness of MAT is enhanced as a result of keeping the nodes of the filial sets together in secondary storage pages.

## 6.5   Comparison of Performance

In this section, we compare the performance of the MAT data structure with other multidimensional data structures. The basis for the comparison is the query, preprocessing, and storage costs when the directory resides on the main memory. The comparison of the MAT with the data structures such as the MDBT, $kB$-tree, BST-complex, etc. was carried out earlier. They all have nearly the same performance measures, as observed in the last sections. Here, we consider the data structures that do not resemble the MAT very closely in the structural aspects. These include the sequential scan, projection, quadtree, $k$-d tree, range tree, non-overlapping $k$-ranges, and overlapping $k$-ranges. An excellent survey of these data structures for the range search can be found in Bentley (1980).

A complete match query is of complexity $O(\log N)$ in the MAT data structure, quadtree, $k$-d tree, etc. Similarly, a partial match query can be answered in $O(N^{1-1/k} \log(N^{1/k}))$ in the MAT. The sequential scan has complexity of $O(N)$ for both the complete and partial match query. The projection data structure has the complexities of $O(t \log tN^{1-1/k})$, and $O(kN^{1-1/k})$ for partial match and complete query respectively. The $k$-d tree and quadtree have the time complexities of $O(\log N)$ and $O(N^{1-1/k})$ for complete and partial match queries. The other data structures such as range trees, d-fold tree, etc. are specifically designed for range queries and, in general, are not well suited for complete and partial match queries.

The comparison of the performance of various data structures is depicted in Table II. Basically, there are four types of data structures from the range search point of view. The first type has complexity of $O(N^{1-1/k} + K_1)$, and the second type has complexity of $O(\log^k + K_1)$. The third and fourth have the range search complexities of $O(\log N + K_1)$, and $O(N^\epsilon)$ respectively. The MAT data structure belongs to the first type. It is to be noted that a reduction in one cost results in an increase in the other costs for any data structure. In the second type of data structure, the reduction in the query cost to $O(\log^k N + K_1)$ is a result of the increases in the preprocessing and storage costs to $O(N \log^{k-1} N)$ and $O(N \log^{k-1} N)$ respectively. This effect is more prominent in the overlapping $k$-ranges. In this data structure the further reduction in the query to $O(\log N + K_1)$ results in the high preprocessing and query costs given by $O(N^{1+\epsilon})$, where $\epsilon$ is a positive number. The inverse effect is observed

TABLE II

COMPARISON OF PERFORMANCE OF SOME MULTIDIMENSIONAL DATA STRUCTURES

| Data structure | Preprocessing cost | Storage cost | Range query cost |
|---|---|---|---|
| Sequential scan | $O(N)$ | $O(N)$ | $O(N)$ |
| Projection | $O(N \log N)$ | $O(N)$ | $O(N^{1-1/k} + K_1)$ |
| Quadtree <br> k-d tree | $O(N \log N)$ | $O(N)$ | $O(N^{1-1/k} + K_1)$ |
| MAT | $O(N \log N)$ | $O(N)$ | $O(N^{1-1/k} \log(N^{1/k}) + K_1)$ |
| Range tree <br> d-fold tree <br> Super B-tree <br> Quintanary tree | $O(N \log^{k-1} N)$ | $O(N \log^{k-1} N)$ | $O(\log^k N + K_1)$ |
| k-fold tree | $O(N \log^{k-1} N)$ | $O(N \log^{k-1} N)$ | $O(\log^{k-1} N + K_1)$ |
| Overlapping <br> k-ranges | $O(N^{1+\epsilon})$ | $O(N^{1+\epsilon})$ | $O(\log N + K_1)$ |
| Non-overlapping <br> k-ranges | $O(N \log N)$ | $O(N)$ | $O(N^\epsilon + K_1)$ |

in non-overlapping $k$-ranges which have the query cost of $O(N^\epsilon + K_1)$ and have preprocessing and storage costs $O(N \log N)$, and $O(N)$ respectively. This seems to suggest an intimate relationship between the query, preprocessing, and storage costs.

When the MAT is organized on secondary storage, its performance should be compared with that of structures such as the grid file, extended hashing, etc. However, the analysis of most of these structures is not carried out in the manner that is done for the MAT. This makes the comparison very difficult.

We close this section by noting that the MAT belongs to the family of data structures that contains $k$-d trees, quadtrees, etc. Despite their similarity in the worst-case complexity, each fares better than the other in the average case of certain aplications. The properties of the problem have to be studied extensively to choose one data structure over the other. The aspects that make the MAT data structure efficient in certain applications are discussed in the next section.

## 7.  Paradigms for the MAT Data Structure

Although the MAT belongs to the class of quadtrees and $k$-d-trees, its worst-case performance can be made better than others in certain applications. The properties of the data and queries can be used to improve the

performance of the MAT in many real-life applications. The data properties such as the sizes of various filial sets, the ordering of attributes, etc, decide the profile of the MAT. This in turn determines the complexities of various queries. The query properties such as the query type, the probability of occurrence of different attributes in a query, the number of records that satisfy the query, etc., decide the performance of different queries on the MAT. This section provides a brief discussion of these two aspects in the form of paradigms.

*Paradigm 1.*   For the same number of records, a MAT with a profile that has fewer large filial sets fares better than a MAT with more smaller filial sets. Figures 12b and 13b give the MAT for the data of Fig. 12a and 13a respectively. In general, data with "large clusters" has a more efficient MAT structure than a MAT with "dispersed" data. For any given data, the profile of
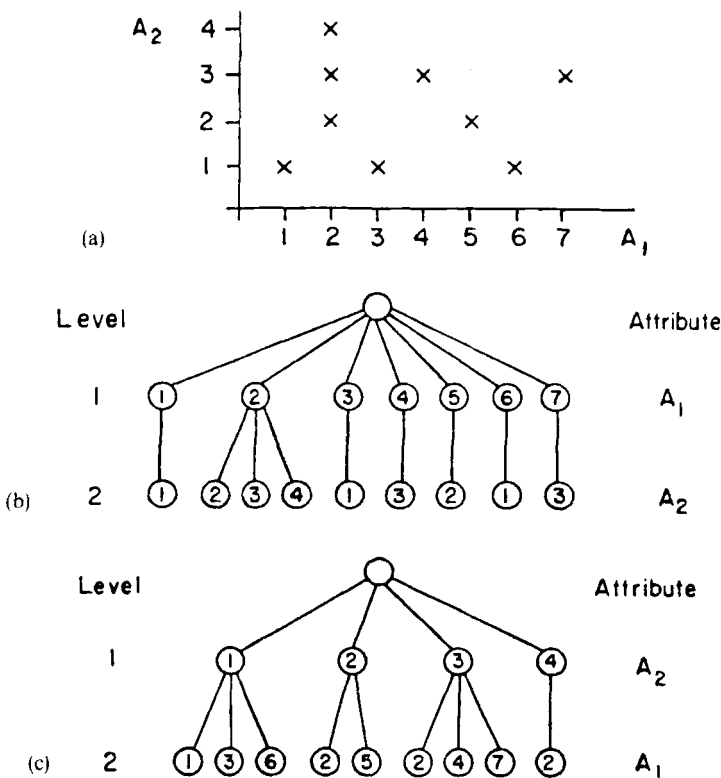


FIG. 12.   (a) Sample data; (b) MAT with attribute ordering $A_1$, $A_2$; (c) MAT with attribute ordering $A_2$, $A_1$.
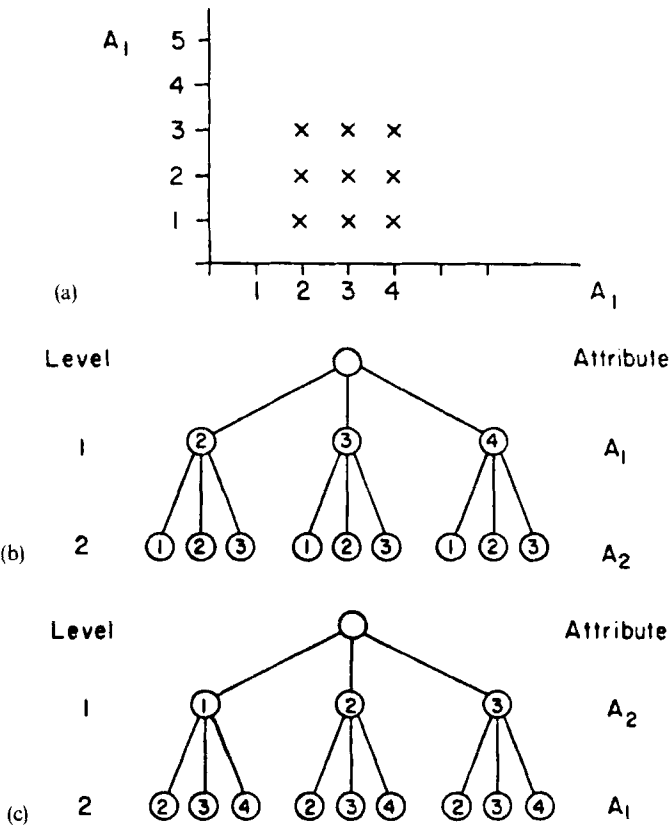
FIG. 13.   (a) Sample data; (b) MAT with attribute ordering $A_1$, $A_2$; (c) MAT with attribute ordering $A_2$, $A_1$.

the MAT depends on the attributes ordering also. Figure 12a illustrates that interchanging the attributes $A_1$ and $A_2$ results in a more compact MAT. The ordering of attributes such that the average filial set size decreases towards the bottom, is shown to be optimal for the average case response of a range query in Rao and Iyengar (1986). Another criterion for ordering the attributes is to minimize the average size of filial sets in all levels. These two requirements may not be met simultaneously. A tradeoff is needed if both the requirements are conflicting.

*Paradigm 2.*   If the more frequently appearing attributes are placed at higher levels, more screening of the nodes will be done at higher levels. This results in fewer nodes to be searched at lower levels. This approach is appropriate for

partial match queries. For a range query, the attributes that are likely to result in fewer qualified nodes may be placed higher in the MAT structure. The growth in the number of nodes to be searched will be restricted as a consequence of this ordering.

The above paradigms suggest MAT as a promising indexed structure for information retrieval applications.

## 8.  Conclusions

The design and analysis of multidimensional data structures and algorithms has gained increasing importance in recent years, especially in the areas of computational geometry, robotics, information retrieval and physical database organization. As a result, a large number of multidimensional data structures have been proposed and studied in various application areas. A brief survey of these existing data structures along with some new paradigms for balancing multidimensional search trees has been presented in this paper. In addition we have explored the potential of Multiple Attribute Trees as a promising multidimensional data structure for application in range query problems. This survey is by no means exhaustive, and omission of any data structure is unintentional.

We have attempted here to illustrate and emphasize the following aspects:

1. The need to develop and analyze simple, efficient balanced self-organizing search tree structures that are useful for real-time applications.
2. A relative analysis of the existing techniques for extension of balanced search trees to multidimensional domains.
3. A model of balanced multidimensional and weighted trees by means of well-defined paradigms.
4. An introductory description of Multiple Attribute Trees and the analysis of their performance for complete match query, partial match query and range query.

It is hoped that the survey will result in a better understanding of the working as well as the development of new performance characteristics of multi-dimensional data structures. In view of the ongoing Fifth Generation Computer Project and the increasing emphasis on the use of high speed computers, a detailed study of the techniques for the design and analysis of balanced self-organizing multidimensional search tree structures becomes quite important.

The notions of time and space complexity are indeed helpful in making an appropriate choice of a data structure in high speed computation. But it is

difficult to analyze the data structure of the computation because of the close interleaving and indeterminacy of mapping costs.

Vaishnavi's [1984, 1987] work described in Section 3 presents a framework for implementations of balanced, self-organizing multidimensional and weighted trees. Such structures need to have efficient worst-case time performance per operation. Multiple attribute trees is a multidimensional data structure with special relevance physical database organization. A major theme of our paper is the exploration of new paradigms for the above structures.

We can view the paradigms discussed here at two different levels: an abstract level and an implementation level.

At the abstract level, the paradigms discussed can be split up into two categories:

1. Strategies to support "structure violations" in the "generalized balanced multidimensional weighted trees".
2. Strategies to exploit structural properties of multiple attribute trees.

At the implementation level, the paradigms for balanced multidimensional weighted trees can be split up into four categories:

1. Structure violations must be "supported" by each of the (next lower dimensional) "sibling" subtrees of the node. We refer to this paradigm as the *local-support strategy* paradigm.
2. In the next strategy, called *global-support strategy*, the structure violations must be supported by each of the "adjoining" subtrees of the nodes causing the structure violation.
3. In the *neighbor-support strategy*, the structure violations must be supported by at least one of the "adjoining" subtrees of the nodes causing the structure violation.
4. By exploiting computational properties of the data structure.

These paradigms have been substantiated with suitable examples. We would like to emphasize here that there is no universal method for designing efficient multidimensional data structures. The trick lies in identifying the right formulation of the computational problem followed by an appropriate decomposition for efficient implementation. The techniques presented in this paper could be used for achieving the above decomposition.

## REFERENCES

Adel'son-Velskij, G. M., and Landis, Y. M. (1962). An algorithm for the Organization of Information. *Soviet Mathematic Doklady* 3, 1259–1263.

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

Alt, H., Mehlhorn, K., and Munro, J. I. (1981). Partial match retrieval in implicit data structures. *Proc. 10th Symp. Math. Foundations of Comp. Sci.*, pp. 156–161.

Avid, Z., and Shamir, E. (1981). A direct solution to range search and related problems for product regions. *Proc. 22nd Annual Symp. on Foundations of Comp. Sci.*, pp. 123–126.

Barnes, M. C., and Collens, D. S. (1973). "Storing Hierarchical Database Structures in Transposed Form." Datafair.

Batory, D. S. (1979). On searching transposed files. *ACM Trans. Database Syst.* 4, 531–544.

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

Bent, S. W. (1982). Dynamic Weighted Data Structures. Ph.D. dissertation, Stanford University, Report No. STAN-CS-82-916.

Bent, S. W., Sleator, D. D., and Tarjan, R. E. (1980). Biased 2-3 Trees. *Proc. 21st Annu. IEEE Symp. on Foundations of Comp. Sci.*, 248–254.

Bent, S. W., Sleator, D. D., and Tarjan, R. E. (1985). Biased Search Trees. *SIAM J. Computing* 14, 545–568.

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

Bentley, J. L. (1977). Solutions to Klee's rectangle problems. Carnegie-Mellon University, Department of Computer Science, typescript.

Bentley, J. L. (1979a). Decomposable searching problems. *Inf. Process. Lett.* 8 (5), 133–136.

Bentley, J. L. (1979b). Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng.* SE-5 (4), 333–340.

Bentley, J. L. (1980). Multidimensional divide-and-conquer. *Commun. ACM* 23 (4), 214–229.

Bentley, J. L., and Burkhard, W. A. (1976). Heuristics for partial match retrieval data base design. *Inf. Process Lett.* 4 (5), 132–135.

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

Bently, J. L., and McGeoch, C. C. (1985). Amortized analysis of self-organizing sequential search heuristics. *Commun. ACM* 28, 404–411.

Bentley, J. L., and Ottmann, T. H. (1979). Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comp.* C-28, 643–647.

Bentley, J. L., and Saxe, J. B. (1979). Algorithms on vector sets. *SIGACT News* 11, 36–39.

Bentley, J. L., and Saxe, J. B. (1980). Decomposable searching problems I: static-to-dynamic transformations. *Journal of Algorithms* 1, 571–577.

Bentley, J. L., and Shamos, M. I. (1977). A problem in multivariate statistics: algorithm, data structure, and application. *Proc. 15th Allerton Conf. on Commun., Control and Comput.*, 193–201.

Bentley, J. L., and Stanat, D. F. (1975). Analysis of range searches in quad trees. *Inf. Process. Lett.* 4 (5), 170–173.

Bolour, A. (1981). Optimal retrieval algorithms for small region queries. *SIAM J. Comput.* 10, 721–741.

Burkhard, W. A. (1983). Interpolation based index maintenance. *Proc. ACM Symp. Principles of Database Systems*, 76–89.

Cardenas, A. F., and Sagamang, J. P. (1974). Double-chained tree data base organization – analysis and strategies. IBM research report RJ 1374.

Casey, R. G. (1973). Design of tree structures for efficient querying. *Commun. ACM* **16** (9), 546–556.

Chang, J. M., and Fu, K. S. (1981). Extended *k*-d tree database organization—a dynamic multiattribute clustering method. *IEEE Trans. Softw. Eng.* **SE-7** (3), 284–290.

Dobkin, D. P., and Lipton, R. (1976). Multidimensional searching problems. *SIAM J. Comput.* **5**, 181–186.

Edelsbrunner, H. (1980a). Dynamic rectangle intersection searching. Technische Universität Graz, Institut für Informationsverarbeitung, Report F47.

Edelsbrunner, H. (1980b). Dynamic data structure for orthogonal intersection queries. Technische Universität Graz Report #59.

Edelsbrunner, H. (1981). A note on dynamic range searching. *Bulletin of EATCS* **15**, 34–40.

Edelsbrunner, H. (1983). A new approach to rectangle intersections. *Int. J. Comput. Math.* **13** (3 & 4), 209–229.

Edelsbrunner, H. (1984). Key-problems and key-methods in computational geometry. *STACS84 Symp. Thero. Aspects of Comp. Sci.*, Paris, April 1984, Lect. Notes Comp. Sci. Springer-Verlag, 1–13.

Edelsbrunner, H., and Maurer, H. A. (1981). On the intersection of orthogonal objects. *Inf. Process. Lett.* **13**, 177–181.

Edelsbrunner, H., and van Leeuwen, J. (1983). Multidimensional data structures and algorithms: Bibliography. IIG, Technische Universität Graz, Austria, Rep. 104.

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. (1979). Extendible hashing—a fast method for dynamic files. *ACM Trans. Database Syst.* **4** (3), 315–344.

Feigenbaum, J. and Tarjan, R. E. (1983). Two new kinds of biased search trees. *Bell Syst. Tech. Journal* **62**, 3139–3158.

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

Fredkin, E. (1960). Trie Memory. *Commun. ACM* **3**, 490–499.

Fredman, M. L. (1981a). The spanning lower bound on the complexity of orthogonal range queries. *J. Algorithms* **1**, 77–87.

Fredman, M. L. (1981b). A lower bound on the complexity of orthogonal range queries. *J. ACM* **28**, 696–705.

Fredman, M. L. (1981). Lower bounds on the complexity of orthogonal range queries. *SIAM J. Comput.* **10**, 1–10.

Gopalakrishna, V., and Veni Madhavan, C. E. (1980). Performance evaluation attribute-based tree organization *ACM Trans. Database. Syst.* **6** (1), 69–87.

Gotlib, C. C., and Gotlib, L. R. (1978). Data types and structures. Prentice-Hall, Englewood Cliffs, N. J.

Gueting R. H. and Kriegel, H. P. (1980). Multidimensional B-tree: an efficient dynamic file structure for exact match queries. *Proc. 10th GI Annu. Conf.*, Informatik Fachberichte, Springer Verlag, 375–388.

Gueting R. H. and Kriegel H. P. (1981). Dynamic *k*-dimensional multiway search under time-varying access frequencies. *Proc. 5th GI Conf. on Theoretical Comp. Sci.*, Lecture Notes in Computer Science 104. Springer-Verlag, New York, 135–145.

Hart, J. H. (1981). Optimal two dimensional range using binary range lists. Tech. Report 76-81, Dept. Computer Science, Univ. Kentucky.

Hirschberg, D. S. (1980). On the complexity of searching a set of vectors. *SIAM J. Comput.* **9**, 126–129.

Horowitz, E., and Sahni, S. (1976). "Fundamentals of data structures." Computer Science Press.

Huddleston, S., and Mehlhorn, K. (1982). A new data structure for representing sorted lists. *Acta Informatica* **17**, 157–184.

Iyengar, S. S., and Raman, V. (1983). Properties and applictions of forest of quadtrees for pictorial information. *BIT* **23**, 472–486.

Jones, L., and Iyengar, S. S. (1984). Space and time efficient virtual quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-6** (2), 244–247.

Kashyap, R. L., and Subas, S. K. C., and Yao, S. B. (1977). Analysis of multiattribute tree database organization. *IEEE Trans. Softw. Eng.* **SE-2** (6), 451–467.

Kirkpatrick, D. G. (1983). Optimal search in planar subdivisions. *SIAM J. Comput.* **12**, 28–35.

Knuth, D. E. (1974). "The art of computer programming, Vol. 3: Sorting and searching." Addison-Wesley, Reading, Mass.

Kosaraju, R. (1981). Localized search in sorted lists. *Proc. 14th Symp. on Theory of Computing*, 62–69.

Kriegel, H. P. (1981). Variants of multidimensional B-trees as dynamic index structure for associative retrieval database systems. *Proc. 7th Conf. Graph Theoretic Concepts in Comput. Sci.*, Linz, Austria.

Kriegel, H. P. (1984). Performance comparison of index structures for multi-key retrieval. *Proc. Annu. Meeting of SIGMOD*, 186–195.

Kriegel, H. P., and Vaishnavi, V. K. (1981a). Multidimensional B$^+$-trees. unpublished manuscript.

Kriegel, H. P., and Vaishnavi, V. K. (1981b). Weighted multidimensional B-trees used as nearly optimal dynamic dictionaries. *Proc. 10th Int. Symp. on Math. Found. of Comput. Sci.* **118**, 410–417.

Kriegel, H. P., and Vaishnavi, V. K. (1981c). A nearly optimal dynamic tree structure for partial-match queries with timevarying frequencies. *Proc. Conf. on Information Sciences and Systems*, 193–197.

Lee, D. T., and Preperata, F. P. (1984). Computational geometry—a survey. *IEEE Trans. Computers* **C-33** (12), 1072–1101.

Lee, D. T., and Wong, C. K. (1977). Worst case analysis of region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica* **9**, 23–29.

Lee, D. T., and Wong, C. K. (1980). Quintary trees: a file structure for multidimensional database systems. *ACM Trans. Database Syst.* **5** (3), 339–353.

Lee, D. T., and Wong, C. K. (1981). Finding intersections of rectangles by range search. *J. Algorithms* **2**, 337–347.

Lien, E. Y., Taylor, E. C., Driscoll, R. J., and Reynolds, L. H. (1975). Binary search tree complex-towards the implementation, *Proc, Very Large Data Base Conf.* Framingham, Mass, 540–542.

Liou, J. H., and Yao, S. B. (1977). Multi-dimensional clustering for data base organization. *Inf. Syst.* **2**, 187–198.

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

Lueker, G. S. (1978). A data structures for orthogonal range queries. *Proc. 19th Annu. Symp. Foundations of Comput. Sci.*, 28–34.

Lueker, G. S. (1979). A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems. Tech. Report #129, Dept. Information and Computer Science, Univ. California, Irvine.

Lueker, G. S. (1982). A data structure for dynamic range queries. *Inf. Proc. Lett.* **15**, 209–213.

Lum, V. Y. (1970). Multi-attribute retrieval with combined indexes. *Commun. ACM* **13**, 660–665.

McCreight, E. M. (1980). Efficient algorithms for enumerating intersecting intervals and rectangles. Xerox Parc Report, CSL-80-09.

Mehlhorn, K. (1979). Dynamic binary search. *SIAM J. Computing* **8**, 175–198.

Mehlhorn, K. (1981). Lower bounds on the efficiency of transforming static data structures to dynamic data structures. *Math. Sys. Theo.* **15**, 1–11.

Mehlhorn, K. (1984a). "Data Structures and Algorithms 1: Sorting and Searching." Springer-Verlag.

Mehlhorn, K. (1984b). "Data structures and algorithms 3: Multidimensional tree structures and computational geometry." Springer-Verlag.

Mehlhorn, K., and Overmars, M. H. (1981). Optimal dynamization of decomposable problems. *Inf. Process. Lett.* **12**, 93–98.

Nievergelt, J., and Reingold, E. M. (1973). Binary search trees of bounded balance. *SIAM J. Comput.* **2**, 33–43.

Nievergelt, J., Hinterberger, H., and Sevcik, K. C. (1984). The grid-file: an adaptable, symmetric multi-key file structures. *ACM Trans. on Database Sys.* **9**, 38–71.

Orenstein. J. A. (1982). Multidimensional TRIEs used for associative searching. *Inform. Proc. Lett.* **4** (4), 150–158.

Ouskel, M., and Scheuermann, P. (1981). Multidimensional B-trees: analysis of dynamic behaviour. *BIT* **21**, 401–418.

Overmars, M. H. (1981). Dynamization of order decomposable set problems. *J. Algorithms* **2**, 245–260.

Overmars, M. H. (1984). "The design of dynamic data structures." *Lect. Notes in Comput. Sci.* **156**, Springer-Verlag, Berlin.

Overmars, M. H., and van Leeuwen, J. (1982). Dynamic multidimensional data structures based on quad- and *k*-d trees. *Acta Informatica* **17**, 267–285.

Pealtz, J. L., Berman, W. J., and Cagley, E. M. (1980). Partial match retrieval using indexed descriptor files. *Commun. ACM* **23**, (9), 522–528.

Preparata, F. P., and Shamos, M. I. (1985). "Computational Geometry." Springer-Verlag, New York.

Rao, N. S. V. (1984). Multiple attribute tree based search and dynamization algorithms. M. E. Thesis, School of Automation, Indian Institute of Science, Bangalore, India.

Rao, N. S. V., and Iyengar, S. S. (1986). Optimal attribute ranking problem in MAT data structure. *Int. J. of Comput. Math.* **21**, 1–12.

Rao, N. S. V., Vaishnavi, V. K. and Iyengar, S. S. (1987). On the dynamization of data structures. Tech. Report #85-031, Dept. of Computer Science, Louisiana State Univ., to appear in *BIT*.

Rao, N. S. V., Veni Madhavan, C. E., and Iyengar, S. S. (1984). Range search and dynamization algorithms in multiple attribute tree. Tech. Report, School of Automation, Indian Institute of Science, Bangalore, India.

Rao, N. S. V., Veni Madhavan, C. E., and Iyengar, S. S. (1985). A comparative study of multiple attribute tree and inverted file structures for large bibliographic files. *Inf. Process. and Mgmt.* **18** (5) 200–215.

Rivest, R. L. (1976). Partial-match retrieval algorithms. *SIAM J. Comput.* **5**, 19–50.

Rothnie, J. B., and Lozano, T. (1974). Attribute based file organization in a paged memory environment. *Commun. ACM* **17** (2). 63–69.

Samet, H. (1983). A quadtree medial axis transform. *Commun. ACM* **26** (9), 680–693.

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

Scheuermann, P. and Ouksel, M. (1982). Multidimensional B-trees for Associative Searching in Database Systems. *Inf. Syst.* **7**, 123–137.

Shamos, M. I. (1978). Computational Geometry. Ph.D. Thesis, Dept. Comput. Sci., Yale Univ.

Shamos, M. I., and Hoey, D. (1975). Closest point problems. *Proc. 16th Annu. IEEE Symp. Foundations of Comput. Sci.*, 151–162.

Sleator, D. D., and Tarjan, R. E. (1983a). Self-adjusting binary search trees. *Proc. 15th Annu. Symp. on Theory of Computing*, 235–245.

Sleator, D. D., and Tarjan, R. E. (1983b). A data structure for dynamic trees. *J. Comput. and Syst. Sci.* **26**, 362–391.

Sleator, D. D., and Tarjan, R. E. (1985). Self-adjusting binary search trees. J. ACM 32(3), 652–686.

Subas, S. K. C. and Kashyap, R. L. (1975). Database organization—an access and storage method. Tech. Report #TREE 75-32, School of Engineering, Purdue Univ.

Sussenguth, E. H., Jr. (1963). Use of tree structures for processing files. *Commun. ACM* **6** (5) 272–279.

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

Tsakalidis, A. K. (1984). An optimal implementation for localized search. Technical Report A 84/06, University of Sarlandes, West Germany.

Vaishnavi, V. K. (1982). Computing point enclosures. *IEEE Trans. Comput.* **C-31** (1), 22–29.

Vaishnavi, V. K. (1983a). On the worst-case efficient implementation of weighted dynamic dictionaries. *Proc. 21st Annu. Allerton Conf. on Communication, Control, and Computing*, 647–655.

Vaishnavi, V. K. (1983b). How to balance a multidimensional search tree. *Proc. 3rd Conf. on Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India.

Vaishnavi, V. K. (1984). Multidimensional height-balanced trees. *IEEE Trans. Comput.* **C-33** (4), 334–343.

Vaishnavi, V. K. (1986). On the height of multidimensional height-balanced trees. *IEEE Trans. Comput.* **C-35**, 773–780.

Vaishnavi, V. K. (1987). Weighted leaf AVL-trees. *SIAM J. Comput.*, 16 (3) 503–537.

Vaishnavi, V. K., and Wood, D. (1980). Data structures for rectangle containment and enclosure problems. *Computer Graphics and Image Process.* **13**, 372–384.

Veni Madhavan, C. E. (1984). Secondary retrieval using tree data structures. *Theor. Comput. Sci.* **33**, 107–116.

Willard, D. E. (1979a). The super B-tree algorithm. Tech. Report TR-03-79, Aiken Computation Lab., Harvard Univ.

Willard, D. E. (1979b). "Predicate oriented database search algorithms." Garland Pub. Company, New York.

Willard, D. E. (1982). Polygon retrieval. *SIAM J. Comput.* 11 (1), 149–165.

Willard, D. E. (1985). New data structure for orthogonal queries. *SIAM J. Comput.* 14 (1) 232–253.

Willard, D. E., and Lueker, G. S. (1985). Adding range restriction capability to dynamic data structures. *J. ACM* 32 (3), 597–617.