Multidimensional Binary Search Trees in Database Applications

JON L. BENTLEY, MEMBER, IEEE

Abstract—The multidimensional binary search tree (abbreviated k-d tree) is a data structure for storing multidimensional records. This structure has been used to solve a number of “geometric” problems in statistics and data analysis. The purposes of this paper are to cast k-d trees in a database framework, to collect the results on k-d trees that have appeared since the structure was introduced, and to show how the basic data structure can be modified to facilitate implementation in large (and very large) databases.

Index Terms—Best match query, binary trees, data structures, k-d trees, multidimensional, partial match query, range query.

I. INTRODUCTION

It is no secret that the designer of a database system faces many difficult problems and is armed with only a few tools for solving them. Among those problems are reliability, protection, integrity, implementation, and choice of query languages. In this paper we will examine a solution to yet another problem that the database designer must face (while keeping the above problems in mind): the design of a database system that facilitates rapid search time in response to a number of different kinds of queries. We will confine our attention to databases of “fixed length records without pointers” specifically; we assume that we must organize a file of N records, each of which contains K keys. Much previous research has been done on problems cast in this framework; the interested reader is referred to Lin, Lee, and Du [1976], Rivest [1976], and Wiederhold [1977] for discussions of many different approaches.

In this paper we will examine a particular data structure, the multidimensional binary search tree, for its suitability as a tool in database implementation. The multidimensional binary search tree (abbreviated k-d tree when the records contain K keys) was introduced by Bentley [1975]. The k-d tree is a natural generalization of the well-known binary search tree to handle the case of a single record having multiple keys. It is a particularly interesting structure from the viewpoint of database design because it is easy to implement and allows a number of different kinds of queries to be answered quite efficiently. The original exposition of k-d trees was cast in geometric terms, and since that time the k-d tree has been used to solve a number of problems in “geometric” databases arising in data analysis and statistics. The purposes of this paper are to cast k-d trees in a database framework, to collect the results on k-d trees that have appeared since the structure was introduced, and to show how the basic data structure can be modified to facilitate implementation in large (and very large) databases.

Since k-d trees are a natural generalization of the standard binary search trees we will review that well-known data structure in Section II. In Section III we develop the k-dimensional binary search tree (k-d tree). We describe how different types of searches can be performed in Section IV and discuss the maintenance of k-d trees in Section V. Section VI faces the problems of implementing k-d trees on different storage media, and a concrete example is then investigated in Section VII. Directions for further work and conclusions are offered in Sections VIII and IX.

II. ONE-DIMENSIONAL BINARY SEARCH TREES

In this section we will briefly review binary search trees; a more thorough exposition of this data structure can be found in Knuth [1973, section 6.2]. Fig. 1(a) is an illustration of a binary search tree representing the numerically-valued keys 31, 41, 15, and 92 (which were inserted in that order). In Fig. 1(b) the additional key 28 has been inserted. The defining property of a binary search tree is that for any node x the key values in the left subtree of x are all less than the key value of x and likewise the key values in the right son are greater than x's. To search to see if a particular value y is currently stored in a tree one starts at the root and compares y to the value of the key stored at the root, which we can call z. If y equals z then we have found it, if y is less than z then our search continues in the left son, and if y is greater than...
Fig. 2. Binary search tree implementations. (a) Homogeneous.
(b) Nonhomogeneous.

Then we continue in the right son. This process continues down the tree until the desired element is located. To insert an element we can apply the searching process until it “falls out” of the tree and then change the last “null pointer” observed to point to the new element.

The abstract binary search tree can be implemented on a computer in many different ways. The most popular representation of a node in a tree is what we will call the homogeneous. In this representation a node consists of a key field (which holds the single key defining the record), left and right son pointers, and additional fields which hold the rest of the data associated with the record. Note that in this approach a node in the tree serves two distinct purposes: representation of a record and direction of a search. These two functions are separated in a nonhomogeneous binary search tree, in which there are two kinds of nodes: internal and external. An internal node contains a key field and left and right son pointers, but no data; all records are held in external nodes which represent sets of records (or perhaps individual records). In nonhomogeneous trees it is important to make the convention that if a search key is equal to the value of an internal node, then the search continues in the right subtree. Homogeneous trees are typically used when the elements of the tree are inserted successively, and nonhomogeneous trees are usually employed when the elements are to be built once-for-all into a perfectly balanced tree. Fig. 2 depicts a set of records stored in the two kinds of trees. A situation in which the nonhomogeneous tree is superior to the homogeneous tree occurs when the records are to be stored on a secondary storage device. In that case the nonhomogeneous tree offers the advantage that entire records do not have to be read into main memory to make a branching decision when only the key is required; we will cover this point in detail in Section VI.

In the above discussion we have alluded to a number of algorithms for performing operations on binary search trees. These operations are usually described only implicitly; we will name them explicitly to facilitate comparison with the analogous operations on k-d trees to be discussed later. Algorithm 1 searches if a record containing a given key is stored in the tree. Knuth [1973, section 6.2.2] shows that this algorithm takes O(log N) expected time if N elements are currently stored in the tree. Algorithm INSERT inserts a new node into a (homogeneous) tree. Its average running time is also logarithmic; if one builds a tree of N elements by using INSERT N times the expected cost of that procedure is O(N log N). An alternative approach is to build a perfectly balanced tree (for every node, the number of right descendants equals the number of left descendants) by algorithm builds, this can be accomplished in O(N log N) worst case time. Note that while a homogeneous tree may be balanced by builds, it is usually the case that a nonhomogeneous tree is built by this algorithm.
Before we generalize this "one-dimensional" binary search tree to become "multidimensional" (that is, deal with several keys per record instead of just one), it is important that we stop for a moment and examine the "philosophy" of binary search trees. These structures perform three tasks at once. Firstly, they store the records of a given set. Secondly, they impose a partition on the data space (that is, they divide the line into segments). Thirdly, binary search trees provide a directory that allows us to locate rapidly the position of a new point in the partition by making a logarithmic number of comparisons. In the next section we will see how these essential features of binary search trees can be captured by a structure that allows a single record to be retrieved by many different search keys.

III. MULTIDIMENSIONAL BINARY SEARCH TREES

The binary search trees of Section II can be used to organize a file of data in which all records contain just one key field and other data fields. If there are many key fields in each record, however, binary search trees are inappropriate because they use only one of the key fields to organize the tree. We will now see how standard binary search trees can be generalized to make use of all of the key fields in a file $F$ of $N$ records of $k$ keys each. A standard binary search tree (discriminates) during an insertion (that is, tells the insertion to proceed right or left) on the basis of one key field. In a multidimensional binary search tree (k-d tree) this discrimination is done on different keys. Specifically, assume that each record in the file has $k$ keys, $K_1, K_2, \ldots, K_k$. On the first level of the tree we choose to go right or left when inserting a new record by comparing the first key ($K_1$) of the new record with the first key of the record stored at the root of the k-d tree (assumed a homogeneous representation). At the second level of the tree we use the second key as the discriminator, and so on to the $k$th level. Finally at the $(k + 1)$st level of the tree we "wrap around" and use the first key as the discriminator again. We illustrate this concept in Fig. 3. Records in that tree each contain two keys: name ($K_1$) and age ($K_2$). Note how every record in the left subtree of the root has a name field less than the root's, and likewise every record in the right subtree has a greater name field. On the second level right subtrees have greater age values.

We can now give a more formal definition of k-d trees. A homogeneous k-d tree is a binary tree in which each record contains $k$ keys, some data fields (possibly), right and left son pointers, and a discriminator which is an integer between 1 and $k$, inclusive. In the most straightforward version of k-d trees all nodes on level $l$ have the same discriminator, namely $(l \mod k) + 1$. The defining property of k-d trees is that for any node $x$ which is a $j$-discriminator, all nodes in the left subtree of $x$ have $K_j$ values less than $x$'s $K_j$ value, and likewise all nodes in the right subtree have greater $K_j$ value. To insert a new record into a k-d tree we start at the root and search down the tree for its position by comparing at each node visited one of the new record's keys with one of the keys of that node, namely the one specified by the discriminator. Bentley [1975] has shown that if a set of $N$ random records are inserted into a k-d tree then it will require approximately $1.386 \ln N$ comparisons to insert the $N$th record, on the average; the expected cost of performing all $N$ insertions is $O(N \ln N)$.

As there are many implementations of one-dimensional binary search trees, so too are there many implementations of k-d trees. The k-d trees that we have described above correspond to the homogeneous binary search trees; it is also possible to define nonhomogeneous k-d trees. Internal nodes in such k-d trees contain only a discriminator (an integer between 1 and $k$), one key value (chosen by the discriminator), and left and right son pointers. All records in nonhomogeneous k-d trees are stored in external nodes or "buckets." (This version of

---

1We say that a field is a key field if a query can refer to it. For example, a set of records might each contain employee number, department number, and salary fields; if queries can refer only to department number and salary then those two fields are keys while employee number is data.

2Random is defined here as all $O(N \ln N)$ permutations of the values of the keys being equally likely. In statistical terms this is saying that to any record, the values of the keys are chosen independently.
k-d trees was originally proposed by Friedman, Bentley, and Finkel [1977]; a similar implementation was later suggested by Willard [1978]. A file of N records can be built into a perfectly balanced nonhomogeneous k-d tree in O(N lg N) time; an algorithm for accomplishing this is given in Bentley [1975] and will be discussed in Section V. We will see that nonhomogeneous trees speed up many types of searches because records in the tree do not have to be examined "on the way down" in a search; only records in well-chosen buckets must be inspected in their entirety. We will also see that nonhomogeneous trees offer substantial advantages in implementations on secondary storage devices.

Another variation among k-d trees is the choice of discriminators; this is appropriate for both homogeneous and nonhomogeneous trees. We originally described a "cyclic" method for choosing discriminators—we cycle in through all k keys. For many kinds of searches one might do better by choosing as discriminator (say) a key in which the data values are particularly well "spread," or by choosing a key which is often specified in queries. Such approaches are examined by Bentley and Burkhard [1976], Friedman, Bentley, and Finkel [1977], and Zelikow [1978].

IV. SEARCHING IN K-D TREES

In the last section we defined k-d trees and mentioned two algorithms for constructing them: by repeated insertion and a "once-for-all" algorithm that produces a perfectly balanced tree. In this section we will examine a number of different algorithms for searching k-d trees, each appropriate for answering a certain kind of query. We will discuss four particular types of searches in detail, and then briefly mention other types of searching possible in k-d trees.

A. Exact Match Queries

The simplest type of query in a file of k-key records is the exact match query: is a specific record (defined by the k keys) in the file? An algorithm for answering such queries is described by Bentley [1975]. The search proceeds down the tree, going right or left by comparing the desired record's key to the discriminator in the node, just as in the insertion algorithm. In the homogeneous version of k-d trees we will either find the record in one pass on a "full scan" of the tree if the record is present. In the nonhomogeneous version we will be directed to a bucket and can then examine the records in that bucket to see if any are the desired. The number of comparisons to accomplish an exact match search is O(N) in the worst case if the tree is perfectly balanced; it is also O(qlg N) on the average for randomly built trees.

B. Partial Match Queries

A more complicated type of query in a multitype file is a "partial match query with k keys specified." An example of such a query might occur in a personnel file: report all employees with length-of-service = 5 and classification = manager, ignoring all other keys in the records. In general we specify values for t of the k keys and ask for all records that have those t values, independent of the other k minus t values. Bentley [1975] describes an algorithm for searching a k-d tree to answer such queries, which we will now sketch. We start the search by visiting the root of the k-d tree. When we visit a node of the k-d tree that discriminates by value we check to see if the value of the tth key is specified in the query; if it is, then we need only visit one of the node's sons (which son is determined by comparing the desired Kj with that node's Kj value). If Kj is not one of the t keys specified, then we must recursively search both sons. Bentley [1975] shows that if t of k keys are specified then the time to do a partial match search in a file of N records is approximately 2^{-t/k} N.

As an example, if four of six keys are specified in a partial match search of one million records, then only approximately 400 records will be examined during the partial match search.

C. Range Queries

In a range query we specify a range of values for each of the k keys, and all records that have every value in the proper ranges are then reported as the answer. For example, we might be interested in querying a student database to find all students with grade point averages between 3.0 and 3.5, parent's income between $12,000 and $20,000, and age between 19 and 21. This problem arises in many applications; Bentley and Friedman [1978a] mention some of those applications and survey the different data structures currently used for solving the problem.

It is easy to answer a range query in a k-d tree; Bentley [1975] describes an algorithm for range searching similar to the partial-match searching algorithm. As we visit a node that is a discriminator we compare the jth value of that node to the jth range of the query. If the range is entirely below the value then the search continues on the left son, if it is entirely above then the search visits the right son, otherwise both sons are recursively searched. Lee and Wong [1977] have analyzed the worst case performance of that algorithm and have established that the time required to perform a range search is never more than O(N^{1/2} + F), where F is the number of points found in the range. Although it is nice to know that things can never get really bad (at least for small k), the average case of searching is much better. Bentley and Stanat [1975] reported results for a data structure very similar to k-d trees that imply that the expected time for range searching in k-d trees is O(N^{1/2} + F); this is indeed the case, on the average, as confirmed by the analysis of Silberschatz [1978a]. It is difficult to analyze the exact performance of range searching because it is so dependent on the "shape" of the particular query, but empirical evidence strongly suggests that k-d trees are very efficient.

D. Best Match Queries

In some database applications we would like to query the database and find that it contains exactly what we are looking for; a builder might hope to find that he has in his warehouse exactly the kind of steel beams he needs for the current project. But often the database will not contain the exact item, and for t < k, if t = k then this is an exact match search and O(N) time is required.
and the user will then have to settle for a similar item. The most similar item to the desired is usually called the "best match" or the "nearest neighbor" to the desired. In information retrieval systems we hope for a book that discusses all ten topics in our list, but we must settle for one (or two) that mentions only eight. Friedman, Bentley, and Finkel [1977] showed how k-d trees can be used to answer such best match queries (where "best" can be defined by many different kinds of "distance functions"). Their algorithm depends on choosing the discriminators in a sophisticated fashion. They showed that the expected amount of work to find the M best matches to a given record is proportional to \( \lg N + M \) in any fixed dimension. Their algorithm was implemented in Fortran for applications in geometric databases and empirical tests showed that their algorithm is orders of magnitude faster than the previous algorithms, for practical problem sizes. Since that time Zuckerman [1978] has analyzed the worst case of nearest neighbor searching in k-d trees and has shown that although any particular search can be rather expensive, if a search for the nearest neighbor of every point in some fixed set is performed then the cost of searching will average to at most \( O(\lg NP) \).

E. Other Queries

The four types of queries we have already investigated are the most commonly discussed queries in fixed-length database applications. Other query types do arise, however, and k-d trees can often be used to answer them. Bentley [1975] gives a procedure that allows k-d trees to answer "intersection" queries, which call for all records satisfying properties that can be tested on a record-by-record basis. The best match algorithm of Section IV-D finds the best match to a particular record; that can be modified to find the best match to a more general description (such as a range, for example). An interesting modification to the basic idea of k-d trees was made by Eastman [1977], who developed a binary tree data structure appropriate for nearest neighbor searching in document retrieval systems.

V. Maintaining k-d Trees

In Section III we defined the k-d tree and in Section IV we described different algorithms for searching k-d trees; in this section we will investigate the problems of maintaining k-d trees. Specifically we will discuss the problems of building a set of records into a k-d tree, inserting a new element into an existing tree, and deleting an existing element from a tree. We will discuss these problems in the two cases of homogeneous and nonhomogeneous trees.

We have already seen the insertion algorithm for homogeneous trees in Section III. We mentioned that a perfectly balanced tree can be built in \( O(N \lg N) \) time; an algorithm to do so is given by Bentley [1975], which we will now sketch for the case of cyclic homogeneous trees. The first step of the algorithm finds the median \( K_i \) value of the entire set (that element greater than one half of the \( K_i \) values and less than the other half). Then let the record corresponding to that element be the root of the entire tree and put the \( N/2 \) elements with lesser \( K_i \) value in the left subtree and the other \( N/2 \) elements in the right subtree. At the next level we find for each of those two subfiles of \( N/2 \) points their \( K_i \) medians, and use those two records as the roots of the two subtrees. This process continues, finding the medians at each level and partitioning around them. If a linear-time algorithm is used to find medians, then this can be accomplished in \( O(N \lg N) \) time. Deletion of a node in a homogeneous k-d tree seems to be a fairly difficult problem. Bentley [1975] gives an algorithm that can delete a node in \( O(N \lg N) \) worst case time. Fortunately, the average running time of that deletion algorithm is much less: \( O(\lg N) \).

The problems of maintaining a nonhomogeneous k-d tree (compared to a homogeneous) seem to be much easier on the average but more difficult when considering the worst case. Recall that there are two types of nodes in a nonhomogeneous k-d tree: internal nodes that contain only discriminators and pointers, and external nodes (or buckets) that contain sets of records. Friedman, Bentley, and Finkel [1975] report that in best match searching the optimal number of records per bucket is about a dozen; this will probably be a reasonable number for many applications. The algorithm that we sketched above for building a homogeneous tree can be applied almost immediately to build a nonhomogeneous tree; its worst case running time is also \( O(N \lg N) \). A good average-case strategy for insertion and deletion in a nonhomogeneous k-d tree is to merely insert the record into or delete the record from the bucket in which it resides; both of these operations can be accomplished in logarithmic time. If the insertions and deletions are scattered almost equally throughout the file then this method will produce very good behavior. If the resulting tree ever becomes too unbalanced for a particular application, then the optimization algorithm could be run again to produce a new optimal tree. (This is especially appealing if there are periods of inactivity in the database, such as at night in a banking system.) Another benefit of nonhomogeneous trees is that if "multiple writers" are used to perform insertions and deletions then they will have to "lock" only the bucket containing the current record (and no nodes higher in the tree).

VI. Implementing k-d Trees

Our discussion of k-d trees so far in this paper has assumed that the cost of going from a node to its son is constant for all nodes, but this is not true in all implementations of k-d trees. In this section we will investigate the problems of implementing k-d trees on various storage devices.

If k-d trees are to be implemented in the main memory of a computer, then either the homogeneous or nonhomogeneous versions will serve well; the nonhomogeneous version is usually the method of choice, however. If in a high-level programming language the nodes in the tree can be implemented as records, and links to and from them are pointers to records. In a fairly low-level programming language (such as Fortran) the nodes of the tree (both internal and external) are usually implemented by sets of arrays representing the various fields of the nodes. There are many general techniques for trees that can be applied to this domain that result in clean and efficient code. For instance, if the tree is known to be totally balanced (as is often the case in a homogeneous tree), then a heap structure
Another technique that enhances k-d trees is the use of a permutation vector. We define a vector \( \text{PERM}[1..n] \), where \( n \) is the number of records in the tree. When the tree is built, the vector \( \text{PERM} \) originally contains pointers to all the records in the tree. We associate with the root of the tree the permutation range \([1..n]\); this implies that pointers to all descendants of the root can be found between positions 1 and \( n \) in \( \text{PERM} \). If we partition at the root around the \( m \)-th record in a certain key, then we will partition the vector \( \text{PERM} \) such that all records in \( \text{PERM}[1..m-1] \) have lesser value in that key, and likewise that all records in \( \text{PERM}[m+1..n] \) have greater values. The permutation range associated with the left son and right son of the root will then be \([1..m-1]\) and \([m+1..n]\) respectively. The resulting permutation vector proves valuable in two contexts. Firstly, it gives a graceful way of keeping track of the records when building the tree. The second application of the permutation vector is in searching the tree. In range searching (and many other searches) we can often determine that all descendants of a particular node lie in the desired range and are therefore to be retrieved without the permutation vector we would need to traverse that subtree to find the relevant records. With the permutation vector, however, we can just iterate through the appropriate range of \( \text{PERM} \) and find pointers there to all the desired records.

If k-d trees are to be implemented on a secondary storage device such as disk then one would probably use nonhomogeneous trees. As an example, assume that we have a file of ten million records, each of five keys. If we allocate ten records in each bucket then there will be one million buckets in the system; this implies that the height of the "internal" part of the tree is twenty, since \( \log_{10} 10,000,000 = 20 \). Because there are too many internal nodes in the tree to store in the main memory, we must store both internal and external nodes on the disk. We will accomplish this by grouping together on the same disk pages internal nodes that are "close" in the tree (see Knuth [1973, section 6.2.4] for the application of this technique to one-dimensional trees). This process is illustrated in Fig. 4. If the discriminators are of reasonable length then compression techniques can be used to store an internal node in (say) ten bytes of storage; this implies that we can store one thousand internal nodes (or ten levels of the tree) on a ten-thousand-byte disk page. Thus there is a distance of only two pages between the root of the tree and any external node, so if the page containing the root is kept in main memory at all times, any record can be accessed in only two page transfers from disk.

There are a few minor observations that can significantly improve the performance of k-d trees implemented on secondary storage devices. We saw above how it is crucial for the internal nodes to require as little space as possible. One means of achieving this space reduction is through key compression. Instead of storing the entire discriminating key in a node, we need only store enough of its first bits to allow us to later test whether to go right or left. For example, if the discriminator in a name field is "Jefferson" it might be sufficient to store only "Je." Likewise, lower in the tree it is probably not necessary to store some of the leading bits of a discriminator.

Another device that can be used to save space on the pages holding internal nodes is the "implicit" binary tree scheme which obviates the need for pointers in defining a binary tree (this is often called a heap). The root of the tree is stored in position 1 of a vector and the left and right sons of node \( i \) are found in locations \( 2i \) and \( 2i + 1 \) respectively. It might be that finding the exact median in building a k-d tree is very expensive. If this is so, then an approximation to the exact median would probably serve just as well as the exact discriminator.

Walke [1976] has given an algorithm that finds approximate medians of large data sets very efficiently; his algorithm should probably be used in such an application. During a search in a k-d tree implemented on secondary storage only a relatively few pages will be kept in main memory at a time, a "least recently used" page replacement algorithm should probably be used to decide which old page to release when reading in a new page.

The above scheme appears very promising for many different applications of k-d trees on secondary storage devices. Though our analysis of the scheme is only for exact match searching, it should also work very well for all of the other search algorithms described in Section IV.

Note the important role that nonhomogeneous k-d trees play in this secondary storage scheme: because the internal nodes are very small compared to the size of the entire records, many of them can reside on one disk page, drastically reducing the required number of disk accesses. A scheme similar to that has been investigated in detail by Silva-Filho [1978]. He uses the k-d tree only as an index, and stores the records in their entirety in a separate data file.

VII. A Concrete Example

In this section we will investigate the application of k-d trees to a student-record database to see how the issues discussed abstractly in previous sections would apply in a concrete example. Many other examples of k-d trees are available to the interested reader. Williams et al. [1975] discuss a PL/I implementation of a database system containing the records of five thousand alumni of the University of North Carolina. Their system supported queries on four key fields (year of birth, year of graduation, geographical code, and alumni name) and was implemented as a homogeneous tree.

---

Footnote: 1For a description of implementation of homogeneous trees on disk storage see Williams et al. [1975].
on disk storage. Bentley [1975] discusses (unimplemented) applications of k-d trees in geographic information retrieval system and in automatic speech recognition systems. Gottlieb and Gotlieb [1978] describe an (unimplemented) k-d tree approach to answering queries in a database of automobiles. Examples of k-d tree implementations in geometric databases can be found in Bentley and Friedman [1978] (for constructing minimum spanning trees), Lauther [1978] (for performing design rule verification for integrated circuits), and Woodward [1977].

Suppose that we are to organize a database of student records. In this file, there is a record for every student containing three keys: date of birth (DOB), grade point average (GPA), and number of units completed (UNITS). Additionally, there will be other student data in each record that we will not mention here. We will organize this data in a homogeneous k-d tree in which the discriminators are chosen cyclically in the pattern GP, DOB, UNITS. A portion of such a k-d tree is shown in Fig. 5.

Fig. 5. Part of the k-d tree for the student database.

The different queries supported by k-d trees might arise in many contexts in this student database. An exact match query (asking if a student existed with given GPA, DOB, and UNITS) would probably not occur. A partial match query, on the other hand, would be just the thing to help prepare the “Dean’s List” by asking for all students with GPA > 4.0. If we wished to prepare an honor roll of all students in their junior year we might ask for a range search with ranges

\[ 3.5 \leq \text{GPA} < 4.0, \]
\[ 60 \leq \text{UNITS} < 90, \]
\[ \text{DOB} \leq \infty. \]

If a psychologist wanted to perform an experiment with a 25-year-old student at the sophomore level with medium grades, then he could ask for the best match in the student database to the hypothetical student with

\[ \text{DOB} = 6/54, \, \text{UNITS} = 45, \, \text{GPA} = 2.2, \]

and the psychologist could choose the distance function by which the closest of other students to this model will be judged.

A k-d tree implementing this student database in internal memory could be structured as a homogeneous or non-homogeneous tree. The homogeneous tree might be the method of choice if the database were particularly active with insertions and deletions. But in most applications the non-homogeneous tree would be preferred. If the data fields of the records are not small enough to allow all of the records to fit into the primary memory of the computer, then two approaches can be taken to solving this problem. We could put all of the records on secondary storage device (such as disk) under the paging scheme discussed in Section VI, and then use an appropriate page replacement algorithm. A second approach would keep the records themselves in internal memory to serve as an index to the complete file, which resides at a direct-access file on disk.

VIII. FURTHER WORK

Although much research has been done on k-d trees since they were introduced in 1975, there are still many further areas that need work. On the practical side it is important that k-d trees be implemented in real database systems to see how the theory relates to practice. Another fascinating problem that needs investigation is methods for choosing discriminator values. Naive k-d trees chose discriminators cyclically, and the k-d trees of Friedman, Bentley, and Finkel [1977] chose as the discriminating key that key with the largest spread in its subspace of the key space. For many database applications, however, it is important to choose as discriminator a key which is used often in queries. Some heuristics proposed by Bentley and Burkhard [1976] for “partial match trees” might be useful in the context of k-d trees.

Perhaps the most outstanding open theoretical problem on k-d trees is that of maintaining dynamic k-d trees. One approach to this problem is to count for each node in the tree the number of left and right sons. If the ratio of sons' weights for any node ever becomes too unbalanced (defined as a parameter of the tree), then the entire subtree rooted at that node is rebuilt. This scheme guarantees that the length of the longest path in the tree is logarithmic, and the total cost of inserting n nodes is at most \( O(n \log n) \). For one-dimensional binary search trees there are a number of more sophisticated balancing schemes that allow insertions and deletions in logarithmic worst case time while ensuring that the tree never becomes unbalanced. It is not known whether or not there exist appropriate “balancing acts” for k-d trees; this problem appears to be very difficult. An alternative approach to the problem of dynamic k-d trees is to use a multiway tree such as those discussed in Knuth [1973, sections 6.2.4]. Another strategy is to maintain a forest of static k-d trees as a dynamic struc-
ture; this approach has been investigated by Bentley [1976] and Wilford [1978].

IX. Conclusions

In this paper we have investigated multidimensional binary search trees from the viewpoint of the database designer. The structure was defined in Section III, and in Section IV we saw that it supports a number of different kinds of queries. This is an especially important feature for database applications; it is essential that different query types be handled and it is most unattractive to have to store different data structures representing the same file. In Section V we saw a number of different maintenance algorithms for k-d trees. The maintenance algorithms for nonhomogeneous k-d trees are particularly simple to code and are very efficient on the average. In Section VI we investigated the implementation of k-d trees and saw that they can indeed be implemented very efficiently. This implies that k-d trees can be used effectively in large and very large databases. A concrete example was investigated in Section VII, and some areas for further research were then described in Section VIII.

This paper represents one of the first attempts to apply k-d trees to the problems that database designers must face. Although we have only scratched the surface of the application of this data structure in this problem domain, it appears that multidimensional binary search trees will be an important addition to the tool bag of the practicing database designer.

References


