Data Structures for Range Searching

JON LOUIS BENTLEY

Departments of Computer Science and Mathematics, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213

JEROME H. FRIEDMAN

Computation Research Group, Stanford Linear Accelerator Center, Stanford, California 94305

Much research has recently been devoted to "multikey" searching problems. In this paper the particular multikey problem of *range searching* is investigated and a number of data structures that have been proposed as solutions to this problem are surveyed. The purposes of this paper are to bring together a collection of widely scattered results, to acquaint the reader with the structures currently available for solving the particular problem of range searching, and to display a set of general methods for attacking multikey searching problems.

Keywords and Phrases: analysis of algorithms, orthogonal range queries, range searching, cells, multidimensional binary search trees, projection

CR Categories. 3.63, 3.74, 5.25

INTRODUCTION

The study of data structures for facilitating rapid searching is a fascinating subject of both practical and theoretical interest. Knuth [KNUT73] provides a definitive treatise on the subject of searching when the search is based on only one "key," but he points out that not much was known at the time his book was published about data structures for sets that have many "keys." This subject area, which is often called "multikey searching," "multidimensional searching," or "multiple attribute retrieval," has been the focus of a great deal of research in the past few years. In this paper we study a small part of this area by surveying the work that has been done on one particular multikey searching problem. This problem is important in itself (having applications in such areas as database systems, statistics, and design automation) and, in addition, serves as a representative of the entire class of multikey searching problems.

We need some definitions to describe this particular searching problem precisely. In database terminology a *file* is a collection of *records*, each containing several *attributes* or *keys*. A *query* asks for all records satisfying certain characteristics. An *orthogonal range query* asks for all records with key values each within specified ranges (that is, each key is between specified upper and lower bounds). The process of retrieving the appropriate records is called *range searching*. This problem can also be cast in geometric terms by regarding the record attributes as coordinates and the k values for each record as representing a point in a k-dimensional coordinate space. The file of records then becomes a point set in k-space. The intersection of the query ranges is a k-dimensional hyperrectangle in the space (that is, a "box"), and a range query calls for finding all points lying inside

This research was supported in part by the Office of Naval Research under Contract N00014-76-C-0370 and in part by the Department of Energy

CONTENTS

this hyperrectangle. We will often cast range searching in this geometric framework as an aid to intuition.

Range searching arises in many applications. In a geographic database of U.S. cities one might seek a list of all those with latitude between 37° and 41° and longitude between 102° and 109° (defining the state of Colorado). To compile an honor list of older students, a university administrator may wish to know those students whose age is between 21 and 24 years and whose grade point average is between 3.5 and 4.0. In data analysis it is often useful to do separate analyses on sets of data lying in different regions (hyperrectangles) of the observation space and then compare (or contrast) the respective results. (At the Stanford Linear Accelerator Center, for example, over 10 hours per week of IBM 370/168 time is devoted to this application.) In statistics, range searching can be employed to determine the empirical probability content of a hyperrectangle, to determine empirical cumulative distributions, and to perform density estimation (see LOFT65). Lauther [LAUT78] describes how range searching can be used to solve a design automation problem in very large-scale integrated circuitry (VLSI).

This paper has been written with two distinct audiences in mind. For the expert in searching (with background either in database systems or theoretical computer science), this paper is intended as a survey that gathers together and presents in a common terminology a number of results that have recently appeared on the problem of range searching. This problem is of particular interest for two reasons: First, it is an important problem in many practical applications (and a difficult theoretical problem!); second, the methods that we investigate are broadly applicable to many other multikey searching problems. The second type of reader for whom this paper is intended is a computer scientist who is somewhat familiar with data structures for single-key searching, and who would like a tutorial on the problem of range searching. For this reader, the methods that we discuss are described on an intuitive level, and references are given to more precise descriptions elsewhere in the literature.

In Section 1 of this paper we examine six data structures for the range searching problem in some detail, and then briefly compare those structures at the end of the section. Additional work (that both has been done and needs to be done) is described in Section 2, and conclusions are then offered in Section 3.

1. THE DATA STRUCTURES

In this section we investigate a number of search methods for range searching. Each search method is specified by a *data structure* for storing the data and algorithms for building (which we call preprocessing) and searching the structure. We will analyze a search structure (say A) by giving three cost functions of N (the number of points) and k (the number of dimensions):

- $P_A(N, k)$, the cost of *preprocessing* N points in k-space into a data structure;
- $S_A(N, k)$, the *storage* required by the data structure;
- $Q_A(N, k)$, the search time or *query* cost.

These costs can be analyzed in terms of their average or their worst case; we usually speak of the worst-case cost, explicitly mentioning the average whenever we employ it. In many applications one may desire various utility operations on data structures, such as insertion and deletion. In this section we ignore this issue, considering only static (unchanging) files; we then return to

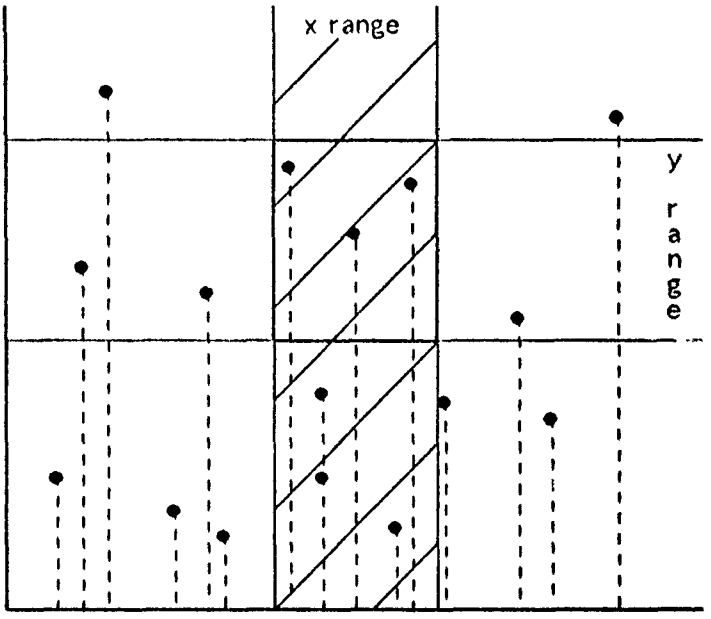


FIGURE 1. Illustration of projection

the question of dynamic structures in Section 2.

1.1 Sequential Scan

The simplest approach to range searching is to store the N points in a sequential list. As each query arrives, all elements of the list are scanned and every record that satisfies the query is reported. If the queries do not have to be handled immediately, then they can be "batched" so that many queries can be processed with one sequential pass through the file. Since all k keys of the N records must be stored and each k-key record is examined as the structure is built or searched, it is easy to see that the sequential scan structure SS has the properties

$$P_{SS}(N, k) = O(Nk),$$
$$S_{SS}(N, k) = O(Nk),$$
$$Q_{SS}(N, k) = O(Nk).$$

Sequential scanning has the advantage of being trivial to implement on any storage medium. It is competitive with the more sophisticated methods described in this paper when the file is small and the number of attributes is large, or when a large fraction of the records in the file satisfy the query (or queries, if they are batched).

1.2 Projection

The projection technique involves keeping, for each attribute, a sequence of the records in the file sorted by that attribute. One can view this geometrically as a projection of the points on each coordinate. The k lists representing the projections can be obtained by using a standard sorting algorithm k times. After preprocessing, a range query can be answered by the following search procedure: Choose one of the attributes, say the ith. Look up the two positions in the ith sequence (using a binary search) of the extreme values defining the range on the ith attribute of the query. All records satisfying the query will be in the list between these two positions just found. This (smaller) list is then searched by brute force. The projection technique is referred to as inverted lists by Knuth [KNUT73]. This technique was applied by Friedman, Baskett, and Shustek [FRIE75] in their solution of the "nearest neighbor" problem and by Lee, Chin, and Chang [LEEC76] to a number of database problems.

The projection technique is illustrated in Figure 1. The points represent a set of sixteen records of two keys each, represented by the x- and y-coordinates. The dashed lines are the projection of the rec-

ords onto the x-coordinate (that is, the records sorted into x-order). The vertical slab is the x-range of the query, the horizontal slab is the y-range, and the rectangle that is their intersection contains those points which satisfy the query. To answer this query, we need only investigate the six points that are inside the vertical slab marked by the 45° lines.

One can apply the projection technique with only one sorted list (projection). If the distribution of values of the various attributes is more or less uniform over similar ranges and the query ranges of each attribute are similar, then one list is sufficient. If this is not the case, however, then keeping several lists can often lead to substantial reductions in the query time. The multiple projections are exploited by performing two binary searches in each to find the lower and upper bounds of the respective range, and then searching that projection with the smallest number of records in the range.

The cost analysis of projection is straightforward. To preprocess a file of N records of k keys each, we must perform k sorts of N elements. To store such a file, we must store k lists of N elements each. These facts immediately yield

$$P_p(N, k) = O(kN \log N),$$
$$S_p(N, k) = O(kN).$$

Friedman, Baskett, and Shustek [FRIE75] show that for searches that have almost cubical query regions and find a small number of records (and are therefore similar to nearest neighbor searches), the query time of projection is given by

$$Q_p(N, k) = O(N^{1-1/k}) \quad \text{(average case)}$$

when the point set is drawn from a smooth underlying distribution. The projection technique is most effective when the queries almost always contain one range that excludes most of the file.

1.3 Cells

There are two ways they can search [for the murder weapon] from the body outward in a spiral, or divide the room up into squares—that's the grid method.

From the CBS series *Kojak*,
"Death Is Not a Passing Grade"

Cartographers as well as detectives use the grid (or cell) method. Street maps of metropolitan areas are often printed in the form of books. The first page of the book shows the entire area, and the remaining pages are detailed maps of (say) one-mile-square regions. To find (for example) all schools in a specified rectangle, one would look at the first page to find which squares overlap the rectangle and then check only on those pages of the book to find the schools. This approach can be mechanized immediately. A square of the map corresponds to a cell in k-space, and the points of the file within the cell are stored together in an implementation. The first page of the map book corresponds to a directory that allows one to take a hyperrectangle and look up the set of cells.

The cell technique is illustrated in Figure 2. The sixteen points in that figure represent sixteen records containing two keys each. The points in each cell are stored together in an implementation. The query is given by the rectangle in the upper part of the figure, and to answer it, only those points in the four dashed cells need be investigated. The squares in that figure are the "directory" corresponding to the first page of the map book.

The directory can be implemented in two ways. If the points are (say) uniformly distributed on $[0, 10]^2$ and we have chosen 1×1 cells, then we can use a two-dimensional array as the directory, named DIRECT (0 . . 9, 0 . . 9). In DIRECT (i, j) we would keep a pointer to a list of all points in the cell $[i, i + 1] \times [j, j + 1]$. If we wanted to find all points in $[5.2, 6.3] \times [1.2, 3.4]$, then we would only have to examine cells (5, 1), (5, 2), (5, 3), (6, 1), (6, 2), and (6, 3)—we call this "translating" from a range query to a set of cell id's. The multidimensional array works very well when the points are known a priori to be uniformly distributed over some given rectangle in the key space. When this is not known to be the case, one would probably use a search method, such as hashing, for the directory. In this method we name each cell as before; so cell (i, j) is a pointer to the points in $[i, i + 1] \times [j, j + 1]$. Instead of storing all cells, however, we store only those cells that actually contain records of the file. To process a query, we translate the rectangle into a set of cell id's (as we did above), look

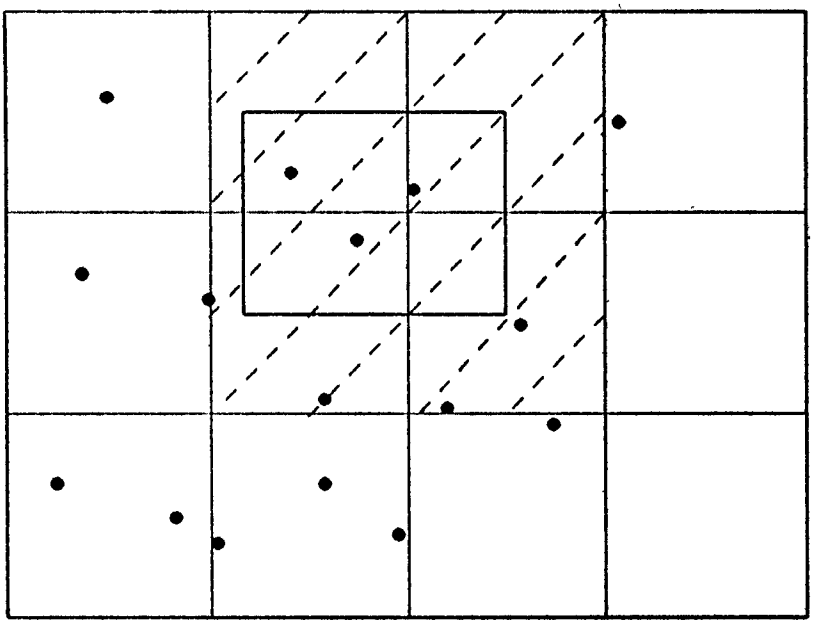


FIGURE 2 Illustration of cells

up those id's, and check all the points in the occupied cells for inclusion in the rectangle. The storage required for the cell technique is the storage for the directory plus locations for the linked list representing points in cells; the size of the directory is usually much smaller than N.

Knuth [KNUT73] has discussed this scheme for the two-dimensional case. Levinthal [LEVI66] used a cell technique in three-dimensional Euclidean space for determining all atoms within 5 angstroms of every atom in a protein molecule—he referred to this as "cubing." The idea of using hashing for the cell directory was first described by Yuval [YUVA75], and was later used by Rabin [RABI76] to solve the "closest pair" problem. Bentley, Stanat, and Williams [BENT77] discuss a number of different implementations for the directory (two of which we have seen).

The basic parameters of the cell technique are the size and shape of each cell. In analyzing a search there are two costs to count: *cell accesses* (the number of directory look-ups) and *inclusion tests* (testing whether a point satisfies the range query). If the cell size is extremely large, there will be few cell accesses and many inclusion tests. If the cell size is very small, on the other hand, there will be very many cell accesses and very few inclusion tests. Clearly, either extreme is to be avoided.

The best cell size and shape depend on the size and shape of the query hyperrectangle. Bentley, Stanat, and Williams [BENT77] show that if the query hyperrectangles have constant size and shape so that only their location (in the coordinate space) is unspecified, then for a single grid a nearly optimum size and shape for the cells are the same as those of the query hyperrectangle. For this case the number of cells accessed is 2^k, and the expected search time is proportional to 2^k times the number of points in the range. In this context the performance of cells is given by

$$P_c(N, k) = O(Nk),$$
$$S_c(N, k) = O(Nk),$$
$$Q_c(N, k) = O(2^k F) \quad \text{(average)},$$

where F is the number of records found. In most applications, however, the queries will vary in size and shape as well as in location, so there is little information available for making a good choice of cell size and shape.

1.4 *k*-d Trees

In this section we examine a data structure called the "k-dimensional binary search

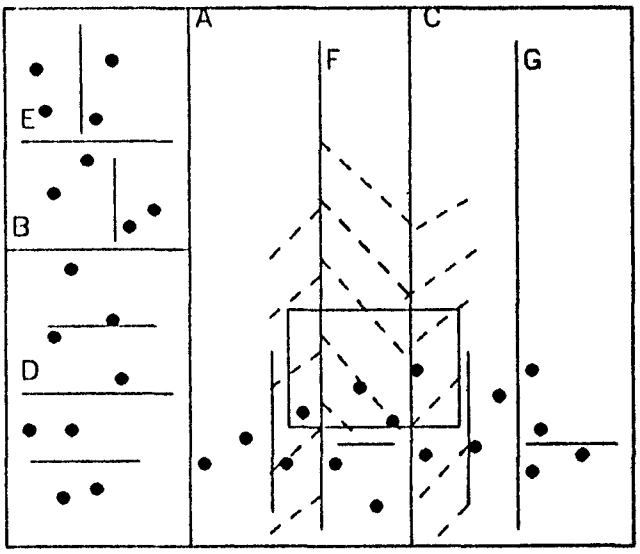
tree," which is usually abbreviated as "k-d tree." This structure is a natural generalization of the standard one-dimensional binary search tree, so we will briefly review a special type of that structure (a complete description of binary trees can be found in KNUT73). To build a file of single-key records into a binary search tree, we choose the median of the set as the *discriminator* value and build all records with key values less than or equal to the discriminator into the left subtree of the root (recursively) and all elements with greater key values into the right subtree. This process continues recursively until there are only a few (say six or less) nodes in the set, at which point we store them as a linked list. Note that no records are stored in the internal nodes of such a binary search tree; they are contained only in the leaf nodes or "buckets" at the bottom of the tree. We can answer a range search in this structure by a recursive algorithm that compares the range to the discriminator of the node it is currently visiting. If the range is entirely to one side or the other of the discriminator, only the appropriate son is searched; otherwise both sons are searched recursively.

The single-key binary search tree performs three functions at once: It stores the records of the file (in the external nodes, or "buckets"), it divides the data space into segments (by choosing the discriminators), and it gives a directory among the segments (the tree structure). We now investigate a multidimensional generalization of the binary search tree that performs these same three functions: storing the records, dividing *space* into *hyperrectangles*, and providing a directory among the hyperrectangles. It accomplishes this by using the same idea as the one-dimensional algorithm with one critical exception: In the one-dimensional tree we only have one key to use as the discriminator; in a multidimensional tree we have to choose *at each internal node* one of k keys to use as a discriminator.

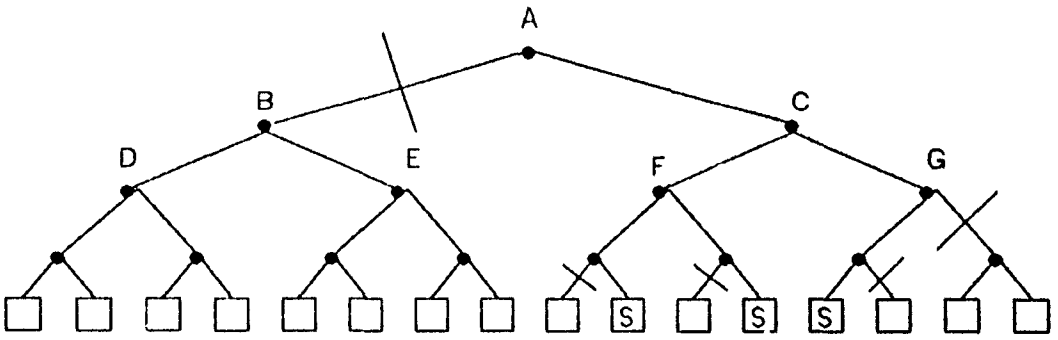
The algorithm for constructing a k-d tree is to choose for the discriminator that coordinate j for which the spread of attribute values (as measured by any convenient statistic, such as variance or distance from minimum to maximum) is maximum for the subcollection represented by the node. The partitioning value is chosen to be the median value of this attribute. This algorithm is then applied recursively to the two subcollections represented by the two sons of the node just partitioned. The partitioning is stopped, creating a terminal node (or bucket), when the cardinality of the subcollection is less than a prespecified maximum, which is a parameter of the procedure. (Friedman, Bentley, and Finkel [FRIE77] found empirically that values ranging from 8 to 16 work well in a Fortran implementation.) The result of this procedure is that the coordinate space is divided into a number of buckets, each containing approximately the same number of points (by the stopping criterion) and each approximately "cubical" in shape (by choosing as discriminator the dimension of maximum spread, which slowly chops long and skinny rectangles into cubes).

Range searching with k-d trees is straightforward. Starting at the root, the k-d tree is recursively searched in the following manner. When visiting a node that discriminates by the jth key (which we call a *j-discriminator*), one compares the jth range of the query with the discriminator value. If the query range is totally above (or below) that value, then one need only search the right subtree (respectively, left) of that node; the other son can be pruned from the search because any node it contains does not satisfy the query in that particular key. If the query range overlaps the node's key (that is, the key is between the low and high bounds of the range), then both sons need be searched. This can be accomplished by searching both sons recursively (the search being implemented by a stack).

The application of k-d trees to (two-dimensional) range searching is illustrated in Figure 3. The k-d tree is depicted in two ways: Figure 3a shows the structure in 2-space, and Figure 3b shows the abstract tree. The root of the tree is internal node A; it is an x-discriminator. The vertical line in the right part of the figure labeled A is the discriminating line. That is, every point to the left of that vertical line is in the left subtree of A (with B as root), and every point to the right is in the subtree with root C. This partitioning continues recursively,



(a)



(b)

FIGURE 3. Illustration of k-d trees a) Planar representation, b) tree representation

and the resulting cells (buckets) in this tree each contain two points. The query rectangle is illustrated in Figure 3a, and the search for all points within the rectangle is illustrated in both figures. The search starts at the root, and since the query rectangle is entirely to the right of the vertical line defined by A, the left subtree of A (with B as root) can be pruned from the search. This is illustrated in Figure 3b by the perpendicular line through the son link from A to B. The search continues, searching both sons of C, both sons of F, and only the left son of G. A total of three buckets are searched; these buckets are dashed in the planar representation and are marked by an *S* in the tree representation.

In the k-d tree as introduced by Bentley [BENT75a], the discriminators are chosen cyclically (that is, the root is discriminated by the first key, its sons by the second, and so on). The idea of "adaptive partitioning" was proposed by Friedman, Bentley, and Finkel [FRIE77] and makes the k-d tree a structure very "sensitive" to the particular file that it represents. The application of k-d trees to a host of problems can be found in BENT79b, GOTL78, and SILV78b.

Analysis of k-d trees for range searching has been considered by several researchers. The work required to construct a k-d tree and its storage requirements (see BENT79b) are

$$P_k(N, k) = O(N \log N),$$
$$S_k(N, k) = O(Nk).$$

The search cost depends on the nature of the query. Lee and Wong [LEEW80] have shown that in the worst case,

$$Q_k(N, k) \leq O(N^{1-1/k} + F)$$

where F is the number of points found in the region. If the query range is almost cubical and the number of records that satisfies the query is small (so that the range query is similar to a nearest neighbor search), then Friedman, Bentley, and Finkel's [FRIE77] analysis shows that

$$Q_k(N, k) = O(\log N + F)$$

(average case for small answer).

For the case where a large fraction of the file satisfies the query, Bentley and Stanat [BENT75b] and Silva-Filho [SILV78a] show that

$$Q_k(N, k) = O(F)$$

(average case for large answer).

The k-d tree structure is most effective in situations where little is known about the nature of the queries or a wide variety of queries are expected. It is also useful if other types of queries (in addition to range queries) are anticipated; many other queries supported by k-d trees are discussed by Bentley [BENT79b].

1.5 Range Trees

A number of very similar structures for range searching (of primarily theoretical rather than practical interest) have recently been described by Lueker [LUEK78], Lee and Wong [LEEW80], and Willard [WILL78a]. In this section we investigate the *range tree*, a structure introduced by Bentley [BENT79a] that is also similar to the former structures. It achieves the best worst-case search time of all the structures we have seen so far in this paper, but has relatively high preprocessing and storage costs. For most applications the high storage will be prohibitive, but the range tree is very interesting from a theoretical viewpoint. Since the range tree is defined recursively in dimension (that is, the k-dimensional structure is defined in terms of the $(k-1)$-dimensional structure), we begin our discussion by looking at a one-dimensional structure and then generalize that structure to higher dimensions.

The simplest structure for one-dimensional range searching is a sorted array. The preprocessing sorts the N elements in ascending order by key. To answer a range query, we do two binary searches to find the positions of the low and high end of the range in the array. After these two positions have been found, we can list all the points in that part of the array as the answer to the range query. (Note that this is precisely the projection method applied to the one-dimensional problem.) For this structure we use linear storage and $O(N \log N)$ preprocessing time. The two binary searches each cost $O(\log N)$, and the cost of listing the points found in the region will, of course, be proportional to the number of such points. Letting F be the number of points found in the region, we have

$$P_r(N, 1) = O(N \log N),$$
$$S_r(N, 1) = O(N),$$
$$Q_r(N, 1) = O(\log N + F).$$

We will now build a two-dimensional range tree, using as a tool the one-dimensional sorted arrays (SA's) we described above. The range tree is similar to the "binary search trees" described by Knuth [KNUT73, Sect. 6.2], so we will use his terminology in our discussions. The range tree is a rooted binary tree in which every node has a left son, a right son, a discriminating value (all nodes in the left subtree have a discriminating value less than the node's), and (unlike a regular binary search tree) every node contains an SA. The root of the range tree contains an SA (sorted by y-coordinate) and has as a discriminating value the median x-value for all points. The left subtree of the root has an SA containing the $N/2$ points with x-value less than median *sorted by y-coordinate*. Similarly, the right son of the root represents the $N/2$ points with x-value greater than the median and has an SA of those points sorted by y-coordinate. This partitioning continues so that i levels away from the root we have 2^i subtrees, each representing $N/2^i$ points contiguous in the x-dimension and each containing an SA of the points sorted by y-coordinate. This partitioning continues for a total of (approximately) $\log N$ levels;

we handle small point sets (say, less than a dozen points) by brute force.

The search algorithm for a range tree is most easily described recursively. Each node in the tree represents a range in the x-dimension from the least x-value contained in the subtree to the greatest. When visiting a node, we compare the x-range of the query to the range of the node, and if the node's range is entirely within the query's, then we search that structure's SA for all points in the query's y-range and return. If the query's range does not wholly contain the node's, then we compare the query's x-range to the node's discriminator value. If the range is entirely below the discriminator, we recursively visit the left subtree; if it is above, we visit the right; and if the range overlaps the discriminator, then we visit both subtrees.

The analysis of the planar tree is rather complicated. Since there are $\log N$ levels in the tree and N points are stored on each level, the total storage required is $O(N \log N)$. The preprocessing can be performed in $O(N \log N)$ time if clever techniques are employed. Analysis shows that at most two SA searches are done on each level of the tree (each of cost approximately $\log N$), so the total cost for a search is $O(\log^2 N)$ plus the time for listing the points in the region. Letting F stand, as before, for the total number of points found in the region we have

$$P_r(N, 2) = O(N \log N),$$
$$S_r(N, 2) = O(N \log N),$$
$$Q_r(N, 2) = O(\log^2 N + F).$$

If we step back for a moment, we can see how we built the structure: We constructed a two-dimensional structure by building a tree of one-dimensional structures. We can perform essentially the same operation to yield a three-dimensional structure: We construct a tree containing two-dimensional structures in the nodes. This process can be continued to yield a structure for k-dimensions, which will be a tree containing $(k - 1)$-dimensional structures. This will yield a structure with performances

$$P_r(N, k) = O(N \log^{k-1} N),$$
$$S_r(N, k) = O(N \log^{k-1} N),$$
$$Q_r(N, k) = O(\log^k N + F).$$

The range tree structure is very interesting from a theoretical viewpoint. The asymptotic search time is very fast, but the amount of storage used is usually prohibitive in practice. Although the application of this structure to practical problems will probably be limited to cases when $k = 2$ or 3, it does provide an important theoretical benchmark. It also gives us an interesting technique (recursion in dimension) that might yield fruit in practice. (Indeed, there are some very interesting relationships between range trees and the k-d trees of Section 1.4.)

1.6 *k*-ranges

The *k-range* is an efficient worst-case structure for range searching introduced by Bentley and Maurer [BENT80b]. They developed two types of k-ranges, overlapping and nonoverlapping. Both of these structures involve storing sets of lists of points sorted by different coordinates; additional dimensions are added recursively, much like the range trees of the last section. Because k-ranges are rather complicated to describe and are of primarily theoretical interest, we will not describe them here but only mention their performance. The overlapping k-ranges can be made to have performance

$$P_o(N, k) = O(N^{1+\epsilon}),$$
$$S_o(N, k) = O(N^{1+\epsilon}),$$
$$Q_o(N, k) = O(\log N + F)$$

for any $\epsilon > 0$. It is pleasing to note that the constants "hidden" in the O's of the above equations are just k/ϵ. Overlapping k-ranges have very efficient retrieval time but somewhat high preprocessing and storage costs; their dual structures, nonoverlapping k-ranges, have very efficient preprocessing and storage costs but increased query times. Their performance is

$$P_n(N, k) = O(N \log N),$$
$$S_n(N, k) = O(N),$$
$$Q_n(N, k) = O(N),$$

for any fixed $\epsilon > 0$. The details of these structures can be found in BENT80b. Although these structures were developed primarily as a theoretical device, they might prove efficient in some implementations

(Their primary drawback is that their space requirements are high, and space is usually a critical resource.)

1.7 Other Structures

In the previous sections we have investigated six structures for the range searching problem that (in the authors' opinion) dominate other structures proposed for this problem. In this section we briefly investigate some of these other structures.

Knuth [KNUT73] points out that the notion of cells can be applied recursively. That is, when one of the cubes has more than some certain number of points, the cube is further divided into subcubes of yet smaller size. This scheme implies a multidimensional tree with multiway branching. In terms of both the partitioning imposed on the space and the ease of implementation, this idea seems to be dominated by a data structure called the quad tree.

The quad tree was first described by Finkel and Bentley [FINK74]. It is a generalization of the standard binary search tree, in which every node has 2^k sons. Bentley and Stanat [BENT75b] analyzed the performance of quad trees for "square" range searches in uniform planar point sets, and Linn [LINN73] discussed the fact that quad trees (which he called "search-sort k trees") have advantages over binary trees when used in a synchronized multiprocessor system. This application aside, however, the quad tree seems to be dominated by the k-d trees of Section 1.4.

A great deal of work has been done recently on multikey searching problems that are similar in flavor to the range searching problem. Dobkin and Lipton [DOBK76] and Bentley [BENT80a] have investigated a number of searching problems defined on sets of points in k-dimensional space. Rivest [RIVE76] provides a number of interesting data structures for answering "partial-match" queries, which are essentially range queries in a file in which the keys assume discrete values. For discussions of efficient search methods in the context of database systems, the reader is referred to such papers as LIOU77, SHNE77, YANG77, and YANG78.

1.8 Comparison of Methods

In Sections 1.1 through 1.6 we have discussed six structures for range searching. The performances of these six structures (seven including the two variants of k-ranges) are summarized in Table 1, which shows the preprocessing, storage, and query costs of each structure. All the functions in that table reflect worst-case costs, except those query costs that are footnoted. For those functions the probabilistic assumptions are described in the notes.

Four of these six structures (sequential scan, projection, cells, and k-d trees) have been presented as providing practical solutions to the range searching problem. For each structure there are situations in which it is clearly superior and other situations where it performs badly. In this section we will mention some of these situations and compare the performance of the four methods.

If the file is small and the number of attributes large, if the file is to be searched only a few times, or if the queries can be batched so that nearly all the records in the file satisfy at least one, then sequential scan

TABLE 1. Performance of Data Structures for Range Searching

Structure	$P(N, k)$	$S(N, k)$	$Q(N, k)$
Sequential scan	$O(N)$	$O(N)$	$O(N)$
Projection	$O(N \log N)$	$O(N)$	$O(N^{1-1/k} + F)^{a(1)}$
Cells	$O(N)$	$O(N)$	$O(F)^{a(2)}$
k-d trees	$O(N \log N)$	$O(N)$	$O(N^{1-1/k} + F)$ $O(\log N + F)^{a(3)}$
Nonoverlapping k-ranges	$O(N \log N)$	$O(N)$	$O(N^{\epsilon} + F)$
Range trees	$O(N \log^{k-1} N)$	$O(N \log^{k-1} N)$	$O(\log^k N + F)$
Overlapping k-ranges	$O(N^{1+\epsilon})$	$O(N^{1+\epsilon})$	$O(\log N + F)$

[a] Query times that indicate average case analysis Probabilistic assumptions are
(1) Smooth data sets—very small query region.
(2) Any data set—cell size equals query size.
(3) Smooth data set.

is the method of choice. In other cases one of the more sophisticated methods is likely to be more efficient. Projection does best when the query range on one of the attributes is usually sufficient to eliminate nearly all the file records. For this case the low overhead of searching this structure allows it to dominate the others. In situations where several or many of the attributes serve to restrict the range query, the projection technique performs relatively poorly.

Both the cell and k-d tree structures are appropriate in situations where the query restricts several of the attributes. If the approximate size and shape of the queries are roughly constant and known in advance, then cells defined by a fixed grid with size and shape similar to those of the expected queries is most advantageous. For queries with sizes and shapes that differ considerably from the design, however, performance can be quite poor.

The k-d tree structure is characterized by its robustness to wildly varying queries. The cell design adapts to the distribution of the attribute values of the file records in the k-dimensional coordinate space. The cells all contain very nearly the same number of records; there are no empty cells. In dense regions there are many cells and a correspondingly fine division of the coordinate space; in sparse regions there is a coarser division with fewer cells. For most applications of range searching that are not characterized in the preceding paragraphs, k-d trees are likely to be the method of choice.

2. ADDITIONAL WORK

Our discussion of the data structures in Section 1 is on a very abstract conceptual level, and we have ignored many problems that arise in actual applications of range searching. In this section we briefly examine some of those problems and the solutions that have been proposed to handle them.

All files that we have discussed so far have been *static*; that is, they represent unchanging files. Many applications, however, require *dynamic* structures, in which insertions and deletions can be made. The sequential scan structure is easy to maintain dynamically, and so is the projection structure using methods for maintaining one-dimensional sorted lists described by Knuth [KNUT73]. The cell technique can support insertions and deletions by merely keeping a linked list of the points in each cell and inserting or deleting the new or old record in the appropriate list. Dynamic k-d trees are a more subtle problem and have been discussed by Bentley [BENT79b] and Willard [WILL78b].

Considerable research remains to be done in the development of heuristics for aiding the search methods we have seen. For example, if the range queries in a seven-dimensional problem almost always involve only two of the attributes, then the design of the structure should involve only those two attributes. Heuristics for detecting these and other similar situations would be very helpful. Techniques described by Bentley and Burkhard [BENT76] might prove useful in such an investigation.

Our discussion of all of the data structures has been for the case in which they are implemented in primary memory. Many applications (particularly databases) inherently involve secondary storage media such as disks and tapes. All the structures of Section 1 can be efficiently implemented on such media.[1]

Several researchers have recently considered an interesting generalization of the range searching problem, which calls for adding a *range restriction* to an existing data structure. That is, we already have some structure for performing a particular type of query, and we want to have the capability of saying "perform that query on all records in which this key lies in that range." Bentley [BENT79a], Lueker [LUEK79], and Willard [WILL78a] have developed a number of *transformations* on data structures that allow one to add the range restriction capability. (These transformations actually led to the discovery of both the range tree and the k-range data structures of Section 1.) Although the storage requirements of the resulting structures seem to be too high to make them of im-

[1] For details of these implementations, the reader is referred to BENT78 which is an earlier version of this paper

mediate practical interest, this approach is a novel attack on the problem of constructing data structures for range searching.

An interesting theoretical problem that could prove to be of practical value is proving lower bounds on the complexity of the range searching problem. Saxe [SAXE79] has investigated this problem using the standard "decision tree" model of concrete complexity theory and has shown that k-ranges have optimal worst-case query times. These k-ranges have very high storage requirements, however; so it would be very desirable to have lower bounds that make stronger statements of the form, "if you only use this much storage and preprocessing, then this is the fastest search time you can have." Fredman [FRED79] has recently made progress in this direction. Another interesting open problem is to show lower bounds on the average complexity, rather than just the worst-case complexity.

3. CONCLUSIONS

In this paper we have investigated a number of data structure for the range searching problem. In 1973 Knuth [KNUT73, p. 554] was able to write that "no really nice data structures seem to exist" for the problem of range searching. In this paper we have tried to show that this situation has changed in the interim, and that these changes can have a substantial impact on both the theory and practice of multikey searching.

REFERENCES

BENT75a BENTLEY, J. L. "Multidimensional binary search trees used for associative searching," *Comm ACM* **18**, 9 (Sept. 1975), 509–517

BENT75b BENTLEY, J. L., AND STANAT, D F. "Analysis of range searches in quad trees," *Inf Process Lett* **3**, 6 (July 1975), 170–173

BENT76 BENTLEY, J L, AND BURKHARD, W A. "Heuristics for partial match retrieval data base design," *Inf Process. Lett* **4**, 5 (Feb 1976), 132–135.

BENT77 BENTLEY, J L., STANAT, D. F., AND WILLIAMS, E. H JR "The complexity of fixed-radius near neighbor searching," *Inf Process. Lett.* **6**, 6 (Dec. 1977), 209–212.

BENT78 BENTLEY, J L., AND FRIEDMAN, J H. *A survey of algorithms and data structures for range searching*, Carnegie-Mellon Computer Science Rep CMU-CS-78-136 and Stanford Linear Accelerator Center Rep SLAC-PUB-2189, preliminary version in *Proc. Computer Science and Statistics: 11th Ann. Symp. on the Interface*, March 1978, pp. 297–307.

BENT79a BENTLEY, J. L. "Decomposable searching problems," *Inf. Process. Lett.* **8**, 5 (June 1979), 133–136.

BENT79b BENTLEY, J. L. "Multidimensional binary search trees in database applications," *IEEE Trans Softw. Eng* **SE-5**, 4 (July 1979), 333–340.

BENT80a BENTLEY, J. L "Multidimensional divide-and-conquer," to appear in *Comm. ACM.*

BENT80b BENTLEY, J. L, AND MAURER, H. A. "Efficient worst-case data structures for range searching," to appear in *Acta Inf.*

DOBK76 DOBKIN, D, AND LIPTON, R. J "Multidimensional searching problems," *SIAM J. Comput.* **5**, 2 (1976), 181–186.

FINK74 FINKEL, R. A, AND BENTLEY, J. L. "Quad trees—a data structure for retrieval on composite keys," *Acta Inf* **4**, 1 (1974), 1–9.

FRED79 FREDMAN, M. "A near optimal data structure for a type of range query problem," in *Proc. 11th ACM Symp. Theory of Computing*, May 1979, pp. 62–66.

FRIE75 FRIEDMAN, J H., BASKETT, F., AND SHUSTEK, L. J "An algorithm for finding nearest neighbors," *IEEE Trans Comput.* **C-24**, 10 (Oct. 1975), 1000–1006.

FRIE77 FRIEDMAN, J H., BENTLEY, J L., AND FINKEL, R. A. "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.* **3**, 3 (Sept. 1977), 209–226.

GOTL78 GOTLIEB, C. C., AND GOTLIEB, L. R *Data types and structures*, Prentice-Hall, Englewood Cliffs, N.J , pp. 357–363

KNUT73 KNUTH, D. E. *The art of computer programming*, vol. 3: *sorting and searching*, Addison-Wesley, Reading, Mass., 1973.

LAUT78 LAUTHER, U. "4-dimensional binary search trees as a means to speed up associative searches in design rule verification of integrated circuits," *J Des. Autom Fault-Tolerant Comput.* **2**, 3 (July 1978), 241–247.

LEEC76 LEE, R. C. T., CHIN, Y. H , AND CHANG, S. C. "Application of principal component analysis to multi-key searching," *IEEE Trans. Softw. Eng.* **SE-2**, 3 (Sept 1976), 185–193.

LEEW78 LEE, D. T , AND WONG, C K. "Worst-case analysis for region and partial region searches in multidimensional binary search trees and quad trees," *Acta Inf* **9**, 1 (1978), 23–29.

LEEW80 LEE, D. T., AND WONG, C. K. "Quintary trees: a file structure for multidimensional database systems," to appear in *ACM Trans. Database Syst*

LEVI66 LEVINTHAL, C. "Molecular model-building by computer," *Sci Am* **214** (June 1966), 42–52.

LINN73 LINN, J. *General methods for parallel searching*, Tech Rep. 61, Digital Systems Lab , Stanford U., Stanford, Calif , May 1973.

LIOU77 LIOU, J H., AND YAO, S B "Multi-dimensional clustering for data base organization," *Inf. Syst.* **2** (1977), 187–198.

LOFT65 LOFTSGAARDEN, D. O , AND QUESEN-

BERRY, C. P. "A nonparametric density function," *Ann. Math. Stat.* **36** (1965), 1049–1051

LUEK78 LUEKER, G. "A data structure for orthogonal range queries," in *Proc 19th Symp Foundations of Computer Science*, IEEE, Oct. 1978, pp. 28–34

LUEK79 LUEKER, G. "A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems," Tech. Rep. 129, U of California at Irvine, 1979.

RABI76 RABIN, M O "Probabilistic algorithms," in *Agorithms and complexity. new directions and recent results*, J. F Traub (Ed.), Academic Press, New York, 1976, pp. 21–39.

RIVE76 RIVEST, R L. "Partial match retrieval algorithms," *SIAM J Comput.* **5**, 1 (March 1976), 19–50

SAXE79 SAXE, J. B "On the number of range queries in k-space," to appear in *Discrete Appl Math*

SHNE77 SHNEIDERMAN, B. "Reduced combined indexes for efficient multiple attribute retrieval," *Inf. Syst.* **2** (1977), 149–154.

SILV78a SILVA-FILHO, Y. V. *Average case analysis of region search in balanced k-d trees*, Rep., U. of Kent, Canterbury, England, Nov. 1978.

SILV78b SILVA-FILHO, Y V. *Multidimensional search trees as indices of files*, Rep., U of Kent, Canterbury, England, Dec 1978.

WILL78a WILLARD, D. E. *Predicate-oriented database search algorithms*, Rep. TR-20-78, Harvard U. Aiken Lab., 1978.

WILL78b WILLARD, D. E. "Balanced forests of k-d* trees as a dynamic data structure," informative abstract, Harvard U., Boston, Mass , 1978.

YANG77 YANG, C. "Avoiding redundant record accesses in unsorted multilist file organizations," *Inf Syst.* **2** (1977), 155–158.

YANG78 YANG, C. "A class of hybrid list file organizations," *Inf. Syst.* **3** (1978), 49–58.

YUVA75 YUVAL, G. "Finding near neighbors in k-dimensional space," *Inf. Process. Lett.* **3**, 4 (March 1975), 113–114

RECEIVED JANUARY 1979; FINAL REVISION ACCEPTED AUGUST 1979.