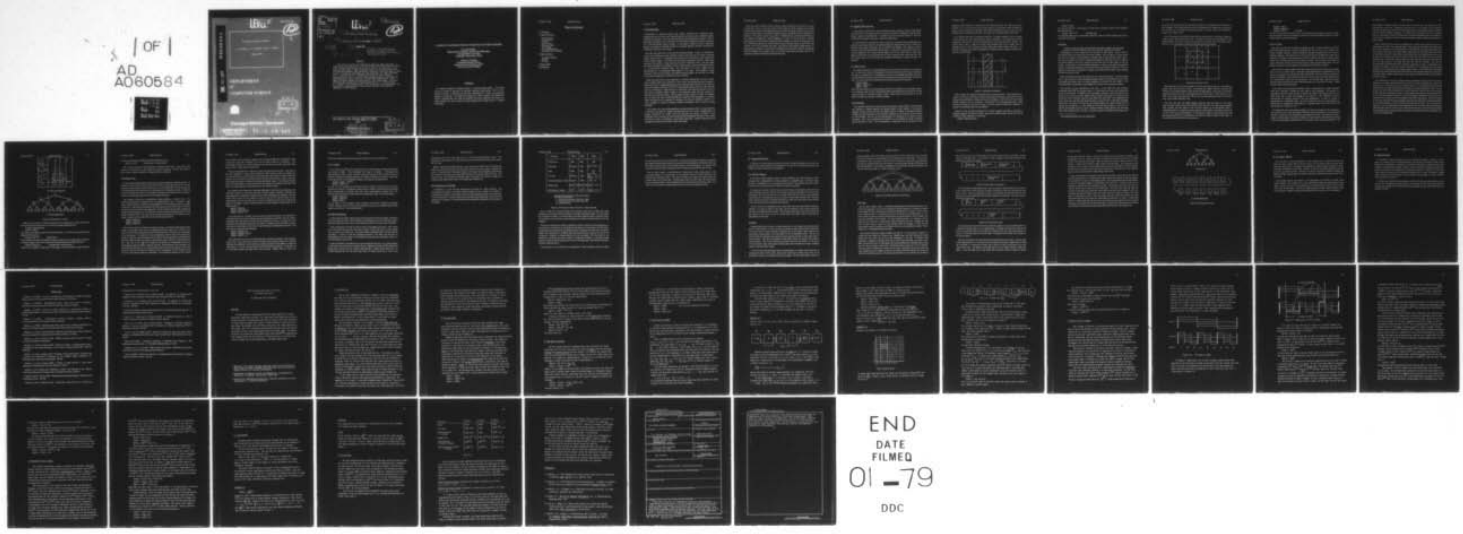
AD-A060 584 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
TWO PAPERS ON RANGE SEARCHING: A SURVEY OF ALGORITHMS AND DATA --ETC(U)
AUG 78 J L BENTLEY, J H FRIEDMAN, H A MAURER N00014-76-C-0370

UNCLASSIFIED CMU-CS-78-136 NL

1 OF 1
AD
A060584



END
DATE
FILMED
01-79
DDC

LEVEL II

CMU-CS-78-136

(12)

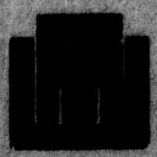
55

Two Papers on Range Searching

J. L. Bentley, J. H. Friedman, and H. A. Maurer

August 1978

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

LEVEL II

(14) CMU-CS-78-136

(12)

(6) Two Papers on Range Searching :

(10) Jon Louis Bentley, Jerome H. Friedman, and H. A. Maurer

(12) 48 p.

(11) August 1978

(9) Interim rept.,

A Survey of Algorithms and Data Structures for Range Searching.
Efficient Worst-Case Data Structures for Range Searching.

Abstract

This report contains two independent papers on range searching. A range search retrieves from a file all records which conjunctively satisfy a set of range requirements for the keys; that is, each key must lie in some specified range. Range searching arises in many applications, such as data base management and statistical computing. The first paper in this report, "A survey of algorithms and data structures for range searching" by J. L. Bentley and J. H. Friedman, describes the known "logical structures" which can be used for range searching and then discusses the implementation of those structures in different storage media. This paper is slanted towards the practitioner. The second paper, "Efficient worst-case data structures for range searching" by J. L. Bentley and H. A. Maurer, is more theoretical. Two new classes of data structures are proposed for range searching, establishing bounds on the asymptotic complexity of the problem.

The research in this paper was supported in part by the Office of Naval Research under contract (15) N00014-76-C-0370.

403 081

A SURVEY OF ALGORITHMS AND 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

See previous page

ABSTRACT

An important problem in database systems is answering queries quickly. This paper surveys a number of algorithms for efficiently answering range queries. First a set of "logical structures" is described and then their implementation in primary and secondary memories is discussed. The algorithms included are of both "practical" and "theoretical" interest. Although some new results are presented, the primary purpose of this paper is to collect together the known results on range searching and to present them in a common terminology.

Table of Contents

1. Introduction

Researchers in database systems have recently identified and investigated many fundamental areas of study in their field; among these are issues such as database security, reliability, and integrity. One area which has not received much attention, however, is that of *algorithmic efficiency*, which is the study of "best possible" algorithms and data structures for answering different kinds of queries. In this paper we apply the tools of *algorithm design and analysis* to database problems by examining algorithms and data structures for answering a particular type of query.

We need some definitions to describe this searching problem. 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. The process of retrieving the appropriate records is called *range searching*. The problem of range searching can be cast in geometric terms. One can regard the record attributes as coordinates, and the k values for each record as representing a point in a k-dimensional coordinate space. The intersection of the query ranges can be represented as a k-dimensional hyperrectangle in this space. The problem of range searching is then to find all points lying inside this hyperrectangle. We will often cast range searching in this geometric framework as an aid to intuition.

Range searching arises in many applications. A university administrator may wish to know those students whose age is between 21 and 24 years and whose grade point average is greater than 3.5. In a geographic database of U.S. cities one might seek a list of all those for which the latitude is between 37° and 41° and longitude between 102° and 109° (defining the state of Colorado). In data analysis it is often useful to do separate analyses on sets of data lying in different regions (ranges) of the observation space and then compare (or contrast) the respective results. (At the Stanford Linear Accelerator Center, for example, over ten 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.

In this paper we survey various algorithms and data structures useful for range searching. In Section 2 we study the "logical" structures and then turn to their implementations in Section 3. Directions for further work and conclusions are offered in Sections 4 and 5. Because this is a survey, we have omitted the more mathematical analyses of the various structures in favor of presenting a more intuitive description. Readers interested in the analyses are referred to the works in which they appear.

There are several problems closely related to range searching on which there has been considerable research. In the future, these methods might be usefully applied to the problem of range searching. Bentley [1975a] discusses the problem of finding all points within a fixed radius of a given point. Yuval [1975] and Bentley, Stanat, and Williams [1977] investigate this problem for the special case of the L_∞ metric. Friedman, Bentley, and Finkel [1977] discuss the problem of finding the k nearest neighbors of a point in a file of N points. Bentley [1976] discusses the problem of finding the nearest neighbor to each of the N points in the file. Domination problems are closely related to range searching; a point is said to dominate another if all of its coordinates are larger. Kung, Luccio, and Preparata [1975] discuss the determination of whether a given point is dominated by any other point. Bentley and Shamos [1977] investigate the calculation of how many points a given point dominates, which is the empirical cumulative distribution evaluated at the point.

2. Logical Structures

In this section we discuss the various methods for range searching in terms of their logical structures; that is, the logical structure of the data at the level of "adjacency" and "pointers" without regard to implementation. In Section 3 we will study the problem of how one implements these logical structures on specific storage media.

A search method is specified by a *data structure* for storing the data and algorithms for building the structure (which we call preprocessing), and searching the structure. There may also be various utility operations such as insertion and deletion. One analyzes a search structure (say S) by giving three cost functions: 1) the cost of *preprocessing* N points in k-space, $P_S(N,k)$; 2) the *storage* required, $S_S(N,k)$; and 3) the search time or *query* cost, $Q_S(N,k)$. These costs can be analyzed in terms of their average or their worst-case cost. We will usually speak of the worst-case cost, explicitly mentioning the average whenever we employ it.

2.1 Brute Force

The simplest approach to range searching is to store each of the N points in a sequential list. As each query arrives all members of the list are scanned and all records that satisfy the query are enumerated. 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. It is easy to see that the brute force structure, B, possesses the properties

$P_B(N,k) = O(Nk)$,
$S_B(N,k) = O(Nk)$, *and*
$Q_B(N,k) = O(Nk)$.

Brute force searching has the advantage of being trivial to implement on any storage medium. It is competitive with the more sophisticated methods described below 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).

2.2 Projection

The projection technique is referred to as inverted lists by Knuth [1973]. This technique was applied by Friedman, Baskett, and Shustek [1976] in their solution of the nearest neighbor problem, and by Lee, Chin, and Chang [1976] to a number of database problems. Projection 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 i-th. Look up the two positions in the i-th sequence (using a binary search) of the extreme values defining the range on the i-th 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 illustrated in Figure 2.1. The points represent a set of sixteen records of two keys each, represented by x and y coordinates. The dashed lines are the projection of the records 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 which is their intersection contains those points which satisfy the query. To answer this query we need only investigate the six points which are inside the vertical slab, marked by the 45° lines.

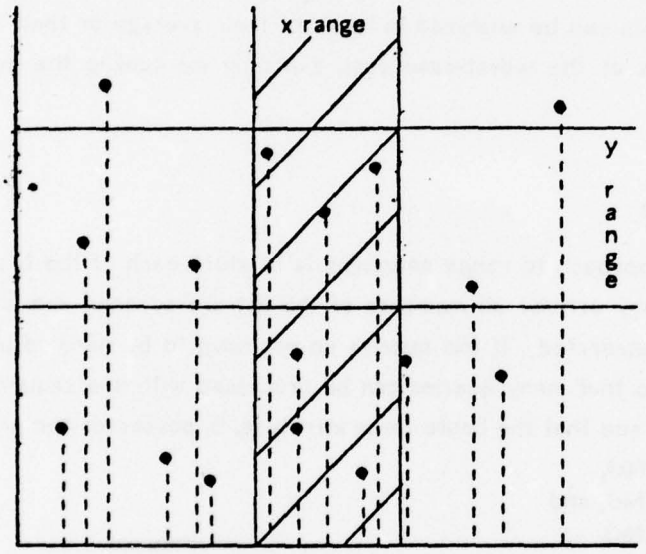


Figure 2.1. Illustration of projection.

One can apply the projection technique with only one sorted list. If the distribution of values of the various attributes are more or less uniform over similar ranges and the query ranges of each attribute are similar, then one list is sufficient. If not, then it can pay to keep sorted sequences on all k attributes. The positions of the corresponding query range extremes are found in each of the k lists. The list for which the difference in positions is smallest is searched between the two positions.

Analysis of the projection technique, P, for nearest neighbor searching is reported in Friedman, Baskett, and Shustek [1976]. Most of this analysis directly carries over to the problem of range searching. It is clear that

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

$S_P(N,k) = O(kN)$.

For searches that find a small number of records (and are therefore similar to near neighbor searches) one has

$Q_P(N,k) = O(N^{1-1/k})$. [Average Case]

The projection technique is most effective when the number of records satisfying each query is usually close to zero.

2.3 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.[1]

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 in the first page to find which squares overlap the rectangle and then check only on those pages 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 as a linked list. The first page of the map book corresponds to a directory which allows one to take a hyperrectangle and look up the set of cells.

Knuth [1973] has discussed this scheme for the two-dimensional case. Levinthal [1966] used a cell technique in three-dimensional Euclidean space for determining all atoms within five angstroms of every atom in a protein molecule--he referred to this as "cubing". Yuval [1975] and Rabin [1976] apply an overlapping cell structure to the closest-pair problem.

The directory can be implemented in two ways. If the points are (say) uniformly distributed on $[0, 10]^2$ and we have chosen 1 x 1 cells, then we can use a two-dimensional array as the directory. 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 then 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), (6,3). The multidimensional array works very well when the points are known *a priori* all to be in some given rectangle. 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 cells which contain points of the file. To process a query we "decode" the rectangle into a

---

[1] From the CBS series *Kojak*, "Death Is Not a Passing Grade".

set of cell id's, look up those id's, and check 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.

The cell technique is illustrated in Figure 2.2. The sixteen points in that figure represent sixteen records containing two keys each. The points in each cell are stored together in an implémentation. 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.

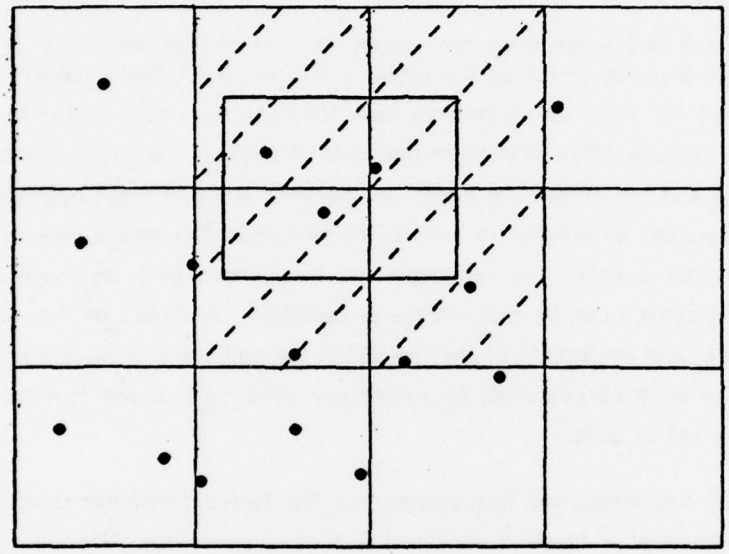


Figure 2.2. Illustration of cells.

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 directory look-ups) and *inclusion tests* (testing whether a point satisfies the range query). If the cells are extremely large, then there will be few cell accesses and many inclusion tests. If the cell size is very small, on the other hand, then there will be very many cell accesses and very few inclusion tests. Clearly either extreme is bad.

The best cell size and shape depends upon the size and shape of the query hyperrectangle. Bentley, Stanat, and Williams [1977] 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 that for 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

$P_C(N,k) = O(Nk)$,
$S_C(N,k) = O(Nk)$, and
$Q_C(N,k) = O(2^k F)$ [Average]

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

2.4 k-d Trees

This data structure was introduced by Bentley [1975b]. Friedman, Bentley and Finkel [1977] introduced adaptive k-d trees and showed that this structure is very effective for nearest neighbor searching. Bentley [1978] has discussed the application of k-d trees to database problems. The application of k-d trees has the effect of dividing the k-space into a collection of irregular hyperrectangles each with the property that they are approximately cubical and all contain nearly the same number of points. This overcomes the problem of empty cells which severely limits the performance of searching with regular grids. The cell pattern induced by k-d trees adapts to the distribution of the points in k-space.

The k-d tree is a generalization of the binary search tree used for sorting and searching. The k-d tree is a binary tree in which each node represents both a subcollection of the points in the space and a partitioning of that subcollection. The root of the tree represents the entire collection. Each nonterminal node has two sons; these son nodes represent the two subcollections defined by the partitioning. The terminal nodes represent mutually exclusive small subsets of the points, which collectively form a partition of k-space. These terminal subsets are called buckets.

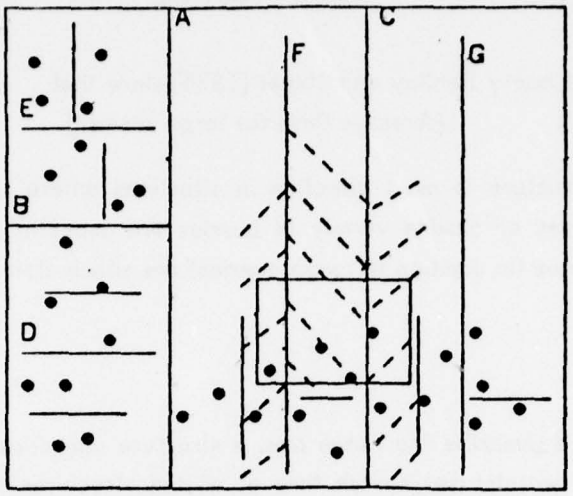
In the case of one-dimensional searching a point is represented by a single coordinate value and a partition is defined by some value of that coordinate. All records in a subcollection with key values less than or equal to the partition value belong to the left son while those with a larger value belong to the right son. That coordinate is thus a *discriminator* for assigning records to the two subcollections. A point in k-space is represented by k coordinate values. Any one of these can serve as a discriminator for partitioning the subcollection represented by a particular node in the tree; that is, the discriminator can range from 1 to k.

The prescription for constructing an adaptive 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) is maximum for the subcollection represented by the node. The partitioning value is chosen to be the median value of this attribute. This prescription is then applied recursively to the two subcollections represented by the two sons of the node just partitioned. The

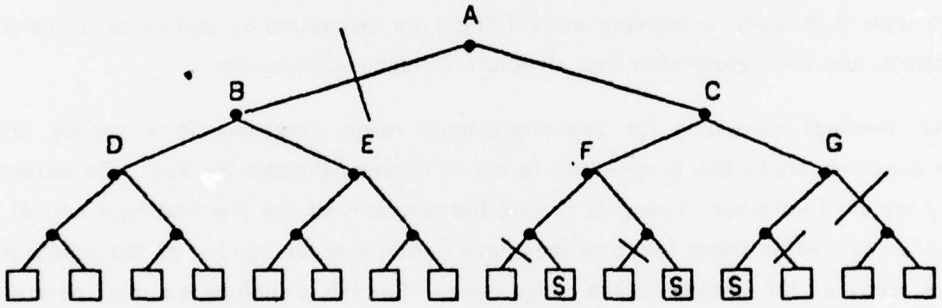
partitioning is stopped, creating a terminal node (or bucket), when the cardinality of the subcollection is less than a prespecified maximum, which is a parameter of the procedure. Friedman, Bentley, and Finkel [1977] found empirically that values ranging from 8 to 16 work well for nearest neighbor searching. 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 criteria) and each approximately "cubical" in shape (by the choice of discriminator).

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 which discriminates by the j-th key (which we call *a j-discriminator*) one compares the j-th range of the query with the j-th key of the node. If the query range is totally above (or below) the key's 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, making use of a stack. A modification can increase the speed of this basic recursive algorithm if it is suspected that many large subtrees will be contained entirely within the query region. The "bounds-array" technique described by Friedman, Bentley, and Finkel [1977] can be employed to detect the inclusion of a subtree within a region, and the points in that region can be listed without the overhead of the tree traversal.

The application of k-d trees to range searching is illustrated in Figure 2.3. The k-d tree is depicted in two ways; Figure 2.3.a shows the structure in 2-space and Figure 2.3.b shows the abstract tree. The root of the tree is internal node A; it is an x-discriminator, and the the vertical line in the right part of the figure labelled A is the discriminating line. That is, every point to the left of that vertical line is in the left subtree of A (which is B), and every point to the right is in the subtree with root C. This partitioning continues recursively, and the cells in this tree each contain two points. The query rectangle is illustrated in Figure 2.3.a, 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 (which is B) can be pruned from the search. This is illustrated in Figure 2.3.b 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.



a.) Planar representation



b.) Tree representation

Figure 2.3. Illustration of k-d trees.

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 are

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

The search cost depends upon the nature of the query. In the very worst case Lee and Wong [1976] show that

$Q_k(N,k) \leq O(N^{1-1/k})$ [Worst Case].

If the number of records that satisfies the query is small so that the range query is similar to a nearest neighbor search then one has from Friedman, Bentley and Finkel [1977] that

$Q_k(N,k) = O(\log N+F)$ [Average Case for small answer]

where F is the number of points found in the region. For the case where a large fraction of

the file satisfies the query Bentley and Stanat [1975] 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. They are also useful if other types of queries (in addition to range queries) are anticipated.

2.5 Range Trees

In this section we describe the *range tree*, a structure introduced by Bentley [1977]. It achieves the best (worst-case) search time of all the structures discussed so far, 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 we will 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 to be 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. For this structure we use linear storage and O(N lg N) preprocessing time. The two binary searches will each cost O(lg 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

$P_R(N,1) = O(N \lg N)$,
$S_R(N,1) = O(N)$, and
$Q_R(N,1) = O(\lg N + F)$.

We will now build a two-dimensional range tree, using as a tool the one-dimensional sorted arrays we described above (which we abbreviate *SA's*). The range tree is similar to the "binary search trees" described by Knuth [1973, Section 6.2] so we will use his terminology in our discussions. The range tree will be 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 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*. Likewise the left 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-coordinate, and each containing an SA of the points sorted by y-coordinate. This partitioning continues for a total of (approximately) lg N levels; we handle small point sets (say less than a dozen ponts) 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. 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. After this 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 somewhat complicated. Since there are lg N levels in the tree and N points are stored on each level, the total storage required is O(N lg N). The preprocessing can performed in O(N lg 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 lg N) so the total cost for a search is $O(\lg^2 N)$ 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

$P_R(N,2) = O(N \lg N)$,
$S_R(N,2) = O(N \lg N)$, and
$Q_R(N,2) = O(\lg^2 N + F)$.

If we step back for a moment we can see how we buiit 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

$P_R(N,k) = O(N \lg^{k-1} N)$,
$S_R(N,k) = O(N \lg^{k-1} N)$, and
$Q_R(N,k) = O(\lg^k N + F)$.

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 probably 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 method that might yield fruit in practice. (Indeed, there are some very

interesting relationships between range trees and the k-d trees of Section 2.4.)

2.6 k-ranges

The *k-range* is an efficient worst-case structure for range searching introduced by Bentley and Maurer [1978]. 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. The overlapping k-ranges can be made to have performance

$$P_O(N,k) = S_O(N,k) = O(N^{1+\epsilon}),$$
$$Q_O(N,k) = O(\lg N + F)$$

for any $\epsilon > 0$. It is pleasing to note that the consants "hidden" in the big-ohs 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, nonoverlapping k-ranges, have very efficient preprocessing and storage costs but increased query times. Their performance is

$$P_N(N,k) = O(N \lg N),$$
$$S_N(N,k) = O(N), \text{ and}$$
$$Q_N(N,k) = O(N^{\epsilon}).$$

for any fixed $\epsilon > 0$. The details of these structures can be found in Bentley and Maurer [1978]. Although these structures were developed primarily as a theoretical device, they might prove efficient in some implementations.

2.7 Other Structures

In this section we briefly mention several structures that we feel are no longer competitive with those discussed above. We include them for completeness and in the hope that someone might be inspired by one of them to invent techniques superior to those we have discussed.

Knuth [1973] 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 the quad tree (see below), which is in turn dominated by the k-d tree.

Finkel and Bentley [1974] describe a structure called the quad tree. It is a generalization of the binary tree in which every node has 2^k sons. Bentley and Stanat [1975] analyzed the performance of quad trees for "square" range searches in uniform planar point sets. Lin [1973] 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 its historical successor, the k-d tree.

Bentley and Shamos [1977] describe a data structure (the ECDF tree) for finding the empirical cumulative distribution of a point (in k-dimensional space) among a collection of points. If only a count of the number of points in the query hyperrectangle is required and, not a listing of the points, then several ECDF searches can be used to obtain that count. This structure has very desirable worst-case query performance but requires storage and preprocessing requirements similar to the range trees of Section 2.5.

2.8 Comparison of Methods

In Sections 2.1 to 2.6 we have discussed six structures for range searching. The performance of these six structures (seven including the two variants of k-ranges) is summarized in Table 2.1, which shows for each the preprocessing, storage, and query costs. All of the functions in that table reflect worst-case costs, except those query costs which are marked with an asterisk. For those functions the probabilistic assumptions are described in the notes.

Structure	P(N,k)	S(N,k)	Q(N,k)
Brute Force	O(N)	O(N)	O(N)
Projection	O(N lg N)	O(N)	$O(N^{1-1/k}+F)$ * [1]
Cells	O(N)	O(N)	O(F) * [2]
k-d trees	O(N lg N)	O(N)	$O(N^{1-1/k}+F)$ [3] O(lg N + F) *
Nonoverlappping k-ranges	O(N lg N)	O(N)	$O(N^{\epsilon}+F)$
Range Trees	$O(N \lg^{k-1} N)$	$O(N \lg^{k-1} N)$	$O(\lg^{k} N + F)$
Overlapping k-ranges	$O(N^{1+\epsilon})$	$O(N^{1+\epsilon})$	O(lg N + F)

*'ed query times indicate average case analysis.
Probabilistic assumptions:

1. Smooth data sets, very small query region.
2. Any data set, cell size equals query size.
3. Smooth data set.

Table 2.1. Performance of data structures for range searching.

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

If the file is small and the number of attributes is large, if the file is to be searched only a few times, or if the queries can be batched so that nearly all of the records in the file satisfy at least one, then brute force is the method of choice. Otherwise one of the other methods is likely to be more efficient. Projection does best when the query range on only one of the attributes is sufficient to eliminate nearly all of 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.

The cell and k-d tree structures are appropriate in those situations where the query

restricts several or many of the attributes. If the approximate size and shape of the queries are roughly constant and are known in advance, then cells defined by a fixed grid with size and shape common to that of the expected queries is most advantageous. For queries with sizes and shapes that differ considerably from the design, however, performance is 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 fine division of the coordinate space; in sparse regions there is a coarser division with fewer cells. If a wide variety of queries are expected then the k-d tree structure should serve best.

3. Implementations

In Section 2 we discussed the various structures for range searching in a more or less abstract way without regard to implementation. We now turn our attention to how one implements these structures on real computers.

3.1 Internal Memory

If the file is small enough so that it can be contained in the internal memory of the computer then implementation of these structures is straightforward. The brute force structure is implemented as a two dimensional (N x k) array. For projection one has k tables of pointers to records; each table is sorted on a different coordinate.

As discussed in Section 2.3, there are two possible ways to associate records with cells when implementing the grid method. If the points are uniformly distributed in a more or less rectangular area (so that there are few empty cells) then the grid can be efficiently represented as a multidimensional array. If there are many empty cells then the k attribute values defining a cell can be treated collectively as a key and a well known search method such as binary searching or hashing can be employed.

The k-d tree can be implemented as any other binary tree; see Knuth [1973] and Bentley [1975b]. It is easy to store for each node a pair of pointers to the records defining the subcollection associated with the node. This facilitates enumeration of the records satisfying the query (if this is the case for all records below that node) without traversing the descendants of that node.

3.2 Disk

Implementing these structures on random access disks is only slightly less straightforward than on central memory. For the most part disk addresses simply replace memory addresses. For brute force one simply performs a sequential scan of the records. With projection the sorted lists contain pointers to the disk address of the corresponding records. The lists for each attribute can themselves be stored on the disk and only one list at a time need reside in central memory. With the cell technique the hash tables contain disk pointers and reside in central memory. The records themselves are stored on disk, with all records in a cell stored on the same disk page. Only pages containing those cells overlapping the query rectangle need be read into central memory.

Tree structures lend themselves nicely to implementation on random access disks; this is discussed by Knuth [1973, p.472]. Figure 3.2.1 shows how the nodes of the tree can be grouped (as shown by the dotted lines) onto disk pages. The size of each page is chosen as

some convenient unit of disk memory (such as a track or sector). While the tree is searched in the usual manner only a few pages at a time need reside in central memory. If the records satisfying the query represent a small fraction of the file then on the order of lg (N/b) disk accesses are required where b is the number of records per page. Bentley [1978] describes this implementation in more detail, and Williams *et al.* [1975] have actually implemented k-d trees for range searching on a random access disk system.

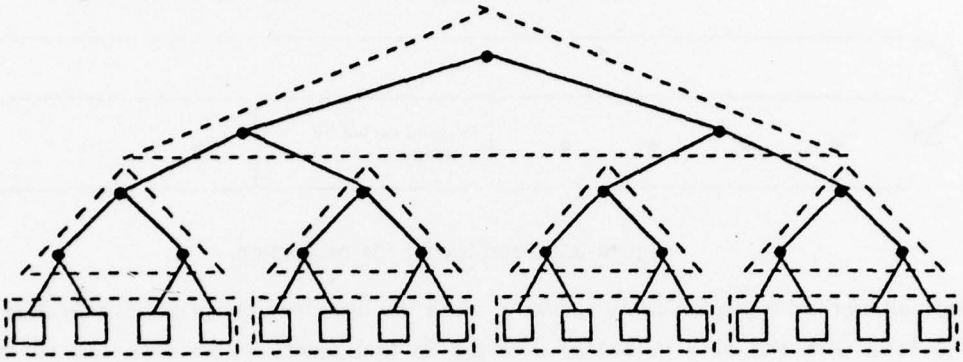


Figure 3.1. Disk pages denoted by dashed lines.

3.3 Tape

By its nature magnetic tape is a sequential storage medium, and therefore ideal for the brute force approach. Even within this sequential limitation, however, it is possible to employ to advantage the other range searching methods described above. In order to read a record from a magnetic tape it is necessary to pass over all records from the beginning to it. It is not necessary, though, to *read* all of those records into central memory or even transmit them from the controller to the channel. On most computing systems it is possible to issue instructions to *skip* one or several blocks without transmitting any data. Although the real time to read a tape is nearly the same whether blocks are skipped or read, the CPU requirement, memory interference, and channel activity can be substantially reduced. This is important in a multiprogramming environment.

The abstract projection method of Section 2.2 calls for storing the set of records in k sorted lists, each sorted by a different key. Magnetic tape is an ideal medium for storing sorted lists--just store each of the k lists sequentially on the tape. In addition some mechanism is needed for deciding which list to search when answering a particular query. When all sorted lists were in main memory this was accomplished by inspecting each; on tape one can store a *sample* of each of the k sorted lists before storing any list in its entirety. This tape layout is illustrated in Figure 3.2. To answer a particular query one counts how many sample records it overlaps in each of the k samples, and then searches that sorted list which has fewest intersections. The search therefore skips over all records until arriving at

the desired sorted list, and then skips over records in that list until we finding a record within the desired range. At that point it starts reading all records and testing to see whether they satisfy all k ranges.

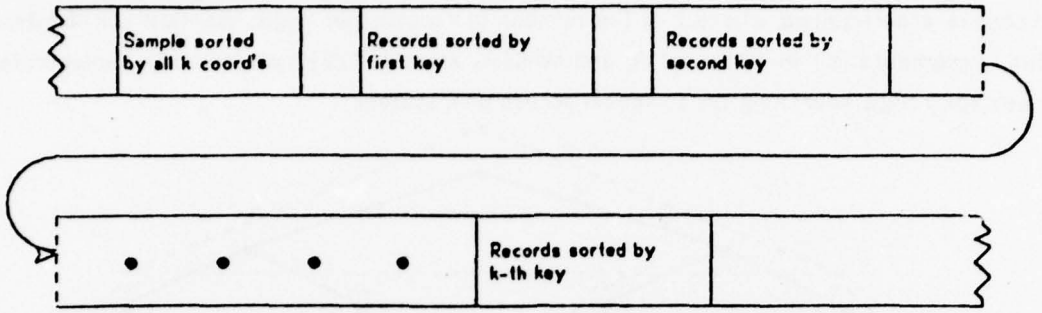


Figure 3.2. Tape layout for projection.

The cell method is implemented similarly. Here the directory comprises the first few blocks of the tape with the data following, arranged so that points within each cell comprise one block of data. The cells overlapping the query are determined from the directory and then those cells are read sequentially from the tape, skipping unwanted blocks. This tape layout is illustrated in Figure 3.3.

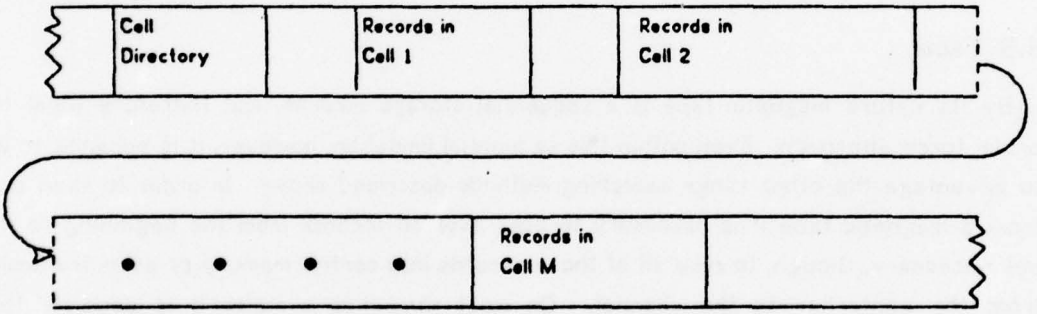


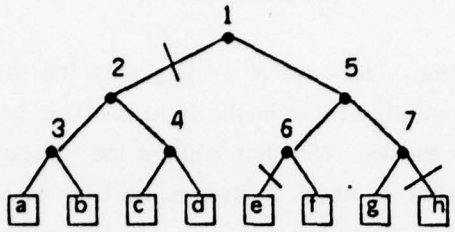
Figure 3.3. Tape layout for cells.

The hierarchical nature of k-d trees and range trees allows for a natural implementation on a sequential storage medium such as magnetic tape. The nodes of the tree are stored in the order of a preorder (node, left son, right son) traversal of the tree. Each node comprises a record. The terminal nodes are the data blocks. Associated with each node is the number, D, of its descendants.

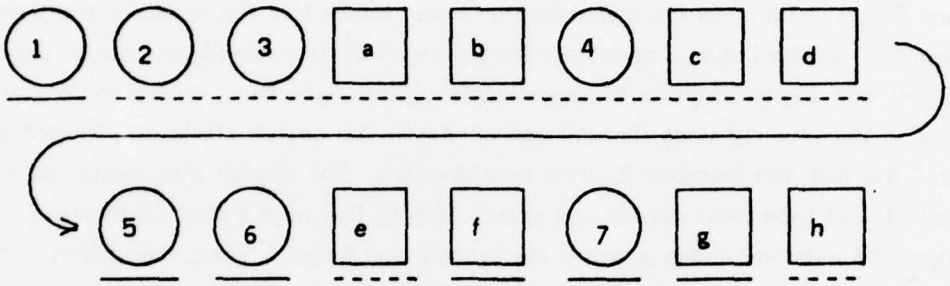
With this arrangement the tree search can proceed directly from the tape. At each node visited (beginning with the root which is the first record on the tape) a determination is made as to whether it is necessary to search one or both of its sons; the outcome of this test yields three cases. The easiest is when both sons are to be visited--continue reading the tape. If only the right son is to be visited, this is also easy--skip the number of blocks

occuppied by the left subtree. The case of visiting only the left subtree is slightly more complicated--stack the number of records in the right subtree, and when control is returned to this node skip that many blocks. With this method the number of blocks read into main memory is equal to the number of nodes visited in the tree search. This technique can be applied to a wide variety of tree searches on tape and the same behavior will be obtained. In particular this method can be used with the range trees of Section 2.5, in that magnetic tape can often accommodate their large storage requirements.

Figure 3.4 illustrates the process of tree searching on tape. Figure 3.4.a is the abstract tree and Figure 3.4.b is its implementation on tape. Notice that the nodes of the tree appear in "preorder" on the tape: a node appears before all of its descendants, and all descendants of the left son appear before all descendants of the right son. A search in the tree is depicted in the tree by lines through son pointers: the search starts at the root and then "prunes" the left son (number 2) from consideration. The search then visits the right son (number 5), and then must recursively search both of that node's sons (numbers 6 and 7); at that point the external nodes e and h are pruned, and nodes f and g are visited. This same search is described on tape in Figure 3.4; nodes which are visited are underlined by solid lines and those which are skipped are underlined by dashed lines. So node 1 is visted, and when it is determined that the left son can be pruned from the search the seven records following it are skipped, and the search proceeds to node 5. Notice that the nodes underlined by solid lines on the tape are exactly those visited by the search in the abstract tree.



a.) Binary tree



b.) Corresponding tape

Figure 3.4. Tape layout for trees.

4. Further Work

Our discussion of range searching has in many respects just scratched the surface and there are many avenues open for further research. All files that we have discussed so far have been *static*, that is, unchanging. Many applications require *dynamic* structures, in which insertions and deletions can be made. Dynamic versions of brute force, projection, and cell structures are easily obtained. Dynamic k-d trees are discussed by Bentley [1975b] and Bentley [1978]. Considerable work remains to be done in the dynamic analysis of all of these structures.

Considerable research also remains in the development of heuristics for aiding these search methods. For example, if in a seven dimensional problem the range queries almost always involve only two of the attributes, then the design of the structure should involve only these two attributes. Heuristics for detecting these and other similar situations would be very helpful. Bentley and Burkhard [1976] might prove useful in such an investigation.

5. Conclusions

The problem of range searching arises in many database applications. In Section 1 of this paper we mentioned some of those applications and defined an "abstract" problem which models the real problems. In Section 2 we used the techniques of "algorithm design and analysis" to describe and analyze a number of data structures for range searching; these abstract structures are interesting from a theoretical viewpoint. In Section 3 we saw how these abstract structures can be efficiently implemented on a number of different storage media, showing that the structures are also practical. Avenues open for further research were mentioned in Section 4.

In 1973 Knuth [1973, 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 database systems.

Bibliography

Bentley, J. L. [1975a]. A survey of techniques for fixed radius near neighbor searching, Stanford Linear Accelerator Center Report SLAC-186, August 1975, 33 pp.

Bentley, J. L. [1975b]. "Multidimensional binary search trees used for associative searching," *Communications of the ACM 18*, 9, September 1975, pp. 509-517.

Bentley, J. L. [1976]. Divide and conquer algorithms for closest-point problems in multidimensional space. Ph.D. Thesis, University of North Carolina, Chapel Hill, North Carolina, 101 pp.

Bentley, J. L. [1977]. Decomposable searching problems, extended abstract, Carnegie-Mellon University Computer Science Department.

Bentley, J. L. [1978]. "Multidimensional binary search trees in database applications," to appear in the Second Computer Software and Applications Conference *Proceedings*.

Bentley, J. L. and W. A. Burkhard [1976]. "Heuristics for partial match retrieval data base design," *Information Processing Letters 4*, 5, February 1976, pp. 132-135.

Bentley, J. L. and H. A. Maurer [1978]. "Efficient worst-case data structures for range searching," submitted for publication.

Bentley, J. L. and M. I. Shamos [1976]. "Divide and conquer in multidimensional space," *Proceedings of the Eighth Symposium on the Theory of Computing*, ACM, May 1976, pp. 220-230.

Bentley, J. L. and M. I. Shamos [1977]. "A problem in multivariate statistics: algorithm, data structure, and applications," *Proceedings of the Fifteenth Allerton Conference on Communication, Control and Computing*, pp. 193-201.

Bentley, J. L. and D. F. Stanat [1975]. "Analysis of range searches in quad trees," *Information Processing Letters 3*, 6, July 1975, pp. 170-173.

Bentley, J. L., D. F. Stanat, and E. H. Williams, Jr. [1977]. "The complexity of near neighbor searching," *Information Processing Letters 6*, 6, December 1977, pp. 209-212.

Dobkin, D. and R. J. Lipton [1976]. "Multidimensional searching problems," *SIAM Journal of Computing 5*, 2, pp. 181-186.

Finkel, R. A. and J. L. Bentley [1974]. "Quad trees--a data structure for retrieval on

composite keys," *Acta Informatica 4*, 1, pp. 1-9.

Friedman, J. H., F. Baskett, and L. J. Shustek [1975]. "An algorithm for finding nearest neighbors," *IEEE Transactions on Computers C-24*, 10, October 1975, pp. 1000-1006.

Friedman, J. H., J. L. Bentley, and R. A. Finkel [1977]. "An algorithm for finding best matches in logarithmic time," *ACM Transactions on Mathematical Software 3*, 3, September 1977, pp. 209-226.

Knuth, D. E. [1973]. *Sorting and Searching, The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading Massachusetts.

Kung, H. T., F. Luccio, and F. P. Preparata [1975]. "On finding the maxima of a set of vectors," *Journal of the ACM 22*, 4, October 1975, pp. 469-476.

Lee, R. C. T., Y. H. Chin, and S. C. Chang [1975]. "Application of principal component analysis to multi-key searching," *IEEE Transactions on Software Engineering SE-2*, 3, September 1976, pp. 185-193.

Lee, D. T. and C. K. Wong [1978]. "Worst-case analysis for region and partial region searches in multidimensional binary search trees and quad trees," *Acta Informatica 9*, 1, pp. 23-29.

Rabin, M. O. [1976]. "Probabilistic algorithms," in *Algorithms and complexity: New directions and recent results*, J. F. Traub (ed.), Academic Press, pp. 21-39.

Williams, E. H. Jr. *et al.* [1976]. PABST Program Logic Manual. Unpublished class project, University of North Carolina, Chapel Hill, North Carolina.

Yuval, G. [1975]. "Finding near neighbors in k-dimensional space," *Information Processing Letters 3*, 4, March 1975, pp. 113-114.

Efficient Worst-Case Data Structures for Range Searching[1]

J.L.Bentley[2] and H.A.Maurer[3]

Abstract

In this paper we investigate the worst-case complexity of range searching: preprocess N points in k-space such that range queries can be answered quickly. A range query asks for all points with each coordinate in some range of values, and arises in many problems in statistics and data bases. We develop three different structures for range searching in this paper. The first structure has absolutely optimal query time (which we prove), but has very high prepocessing and storage costs. The second structure we present has logarithmic query time and $O(N^{1+\epsilon})$ prepocessing and storage costs, for any fixed $\epsilon>0$. Finally we give a structure with linear storage, O(N lg N) prepocessing, and $O(N^{\epsilon})$ query time.

[1] Research in this paper has been supported partially under Office of Naval Research contract N000014-76-C-0373, USA, and by the Austrian Federal Ministry for Science and Research.

[2] Departments of Computer Science and Mathematics, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, USA.

[3] Institut für Informationsverarbeitung, Technical University of Graz, Steyrergasse 17, A-8010 Graz/Austria.

1. Introduction

One of the fundamental problems of computer science is searching, and many efficient algorithms and data structures have been developed for a wide variety of searching problems. Most of these algorithms deal with problems defined by a single search key, however, and very little work has been done on searching problems defined over many keys. Such problems are usually called multi-key or multidimensional (because each of the key spaces can be viewed as a dimension) searching problems. A survey of many multidimensional searching algorithms can be found in Maurer and Ottmann [6]. In this paper we will investigate and (optimally) solve one such multidimensional searching problem.

The problem of interest in this paper is called range searching. Phrased in geometric terms, we are given a set F of N points in k-space to prepocess into a data structure. After we have prepocessed the points we must answer queries which ask for all points x of F such that the first coordinate of x (x_1) is in some range $[L_1,H_1]$, the second coordinate $x_2 \in [L_2,H_2]$, ... , and $x_k \in [L_k,H_k]$. One can also phrase this problem in the terminology of data bases: we are given a file F of N records, each of k keys, to process into a file structure. We must then answer queries asking for all records such that the first key is in some specified range, the second key in a second range, etc. Range searching is called orthogonal range searching by Knuth [4, Sec.6.5.].

Range searching arises in many applications. In purchasing a desk for a certain office we might ask a furniture data base to list all desks of width 80 cm to 120 cm, length 160 cm to 240 cm, and cost $ 100,00 to $ 200.00. Knuth [4, Section 6.5] mentions that range searching arises in geographic data bases: in a file of North American cities we can list all cities in Colorado by asking for cities with latitude in [37^{o}N, 41^{o}N] and longitude in [102^{o}W, 109^{o}W]. Other applications of range searching in statistics and data analysis are mentioned by Bentley and Friedman [3].

In this paper we will study the worst-case complexity of range searching, explicitly ignoring the expected performance of algorithms. The emphasis of this paper is therefore somewhat more "theoretical" than practical. Previous approaches to range searching are discussed in Section 2. In Section 3 we present three new structures for range searching. The first

of these has very rapid retrieval time but requires much storage and preprocessing. The second has slightly increased retrieval time but reduced storage and building costs. The third type of structure is still less efficient as far as query time is concerned, but is optimal in storage requirement and has low prepocessing cost. In Section 4 we prove the optimality of the fast retrieval-time structure of Section 3 by exhibiting a lower bound for range searching. We present conclusions and directions for further research in Section 5.

2. Previous Work

Most of the data structures which have been proposed for range searching have been designed to facilitate rapid average query time. Such structures include inverted lists and multidimensional arrays representing "cells" in the space. These and other "average-case" structures are discussed by Bentley and Friedman [3].

Before we describe existing "worst-case" structures for range searching we must state our methods for analyzing a data structure. Our model for searching is that we are given a set F which we prepocess into a data structure G such that we can quickly answer range queries about F by searching G. Note that all the structures we discuss in this paper are static in the sense that they need not support insertions and deletions. To analyze a particular structure we describe three cost functions as functions of N (the size of F) and k (the dimension of the space). These functions are P(N,k), the prepocessing time required to build the structure G; S(N,k), the storage required by G; and Q(N,k), the time required to answer a query. To illustrate this analysis consider the "brute force" approach to range searching, which stores the N points of the file in a linked list. The prepocessing, storage, and query costs of this structure are all linear in Nk, so the analysis of "brute force" yields

$P(N,k) = O(Nk)$,
$S(N,k) = O(Nk)$, and
$Q(N,k) = O(Nk)$.

The multidimensional binary search tree (abbreviated k-d tree in k-space) proposed by Bentley [1] is a more sophisticated data structures which supports range searches. Bentley showed that the prepocessing and storage costs of the k-d tree are respectively

$$P(N,k) = O(kN \lg N), \text{ and}$$

$$S(N,k) = O(Nk),$$

but he did not analyze the worst-case cost of searching. Lee and Wong [5] later analyzed the query time of k-d trees and showed that it is

$$Q(N,k) = O(kN^{1 - 1/k} + A)$$

where A is the number of answers found in the range.

A second structure for range searching is the range tree of Bentley [2]. This structure is based on the idea of "multidimensional divide-and-conquer" and has performances

$$P(N,k) = O(N \lg^{k-1} N),$$

$$S(N,k) = O(N \lg^{k-1} N), \text{ and}$$

$$Q(N,k) = O(\lg^{k} N + A)$$

for any fixed $k \geq 2$.

3. New Data Structures

In this section we will introduce three new structures for range searching. We will call these data structures k-ranges and consider overlapping and nonoverlapping versions thereof. To simplify our notation we will call overlapping k-ranges just k-ranges but we will always explicitly mention if k-ranges are nonoverlapping. In Section 3.1 we describe (overlapping) k-ranges and establish their performance as

$$Q(N,k) = O(k \lg N + A), \text{ and}$$

$$P(N,k) = S(N,k) = O(N^{2k - 1}),$$

where A is the number of points found. (In Section 4 we will see that this query time is optimal under comparision-based models.) Although k-ranges have very rapid retrieval times they "pay for" this by high prepocessing and storage costs. In section 3.2 we will modify k-ranges to display performance

$$Q(N,k) = O(f(\epsilon) \cdot \lg N) + O(A), \text{ and}$$

$$P(N,k) = Q(N,k) = O(N^{1+\epsilon})$$

for any fixed $\epsilon > 0$.

In Section 3.3 we introduce nonoverlapping k-ranges. Storage and prepocessing costs for this type of data structure are still lower than for the data structures of Section 3.1 and 3.2 (in fact even lower than the ones for range trees of Bentley [2]). However, query time is increased somewhat (and is higher than for range trees). Specifically we show for nonoverlapping range trees a performance of

$Q(N,k) = O(N^{\epsilon})$ ($\epsilon > 0$ can be chosen arbitrarily)

$S(N,k) = O(N)$

$P(N,k) = O(N \lg N)$.

3.1 One level k-ranges

Before describing our data structures and techiques it is convenient to transform the problem of range searching in a k-dimensional set F of N points with arbitrary real coordinates into the problem of range searching in a k-dimensional set $\overline{F}$ of N points with integer coordinates between 1 and N.

Such a "normalization" can be carried out as follows. Let $F_i = \{x_i | x \in F\}$ $(1 \leq i \leq k)$ be the set of numbers occuring as i-th coordinate. For each point $x = (x_1, x_2, \ldots x_k)$ of F take a point $\overline{x} = (\overline{x_1}, \overline{x_2}, \ldots, \overline{x_k})$ into $\overline{F}$, where $\overline{x}_i$ is the "rank" of x_i in F_i. (If the numbers of F_i are sorted into ascending order and duplicates are removed, then the position of x_i in this sequence is it rank). Note that such normalization can be accomplished in time $O(kN \lg N)$ and space $O(N)$. A range query $[L_1, H_1], [L_2, H_2], \ldots, [L_k, H_k]$ in F can be "normalized" into a range query $[\overline{L}_1, \overline{H}_1], [\overline{L}_2, \overline{H}_2], \ldots [\overline{L}_k, \overline{H}_k]$ in $\overline{F}$ in a similar fashion in $2k \lg N$ comparisons.

For the above reasons we can assume in the following that F is a set of N points in k dimensions with all coordinates being integers between 1 and N, and that each range query $[L_1, H_1], \ldots, [L_k, H_k]$ consists of intgers only with

$1 \leq L_i \leq H_i \leq N$ for $i = 1,2, \ldots, k$.

In calculating preprocessing and query times the time required for normalization will of course have to be considered.

Our aim is to store the set F as a k-range, a data structure defined inductively for k = 1,2, ... and permitting the fast processing of range queries. In discussing k-ranges we will first develop the one-dimensional structure and then extend it to successively higher dimensions.

Let G be some subset of F, $G \subseteq F$. To store G as a 1-range we store G as a linear array M of N elements as follows. Each element consists of a set of points M_i and a pointer p_i $(1 \leq i \leq N)$, where M_i is the set of all points of G with first coordinate equal to i, and where p_i points to the "next" nonempty M_j, i.e. to that nonempty set M_j with $i < j$ and j minimal.

Example 3.1

The set {(1,6), (3,3), (5,1), (5,5), (6,2)} stored as a 1-range is shown in Fig. 3.1.

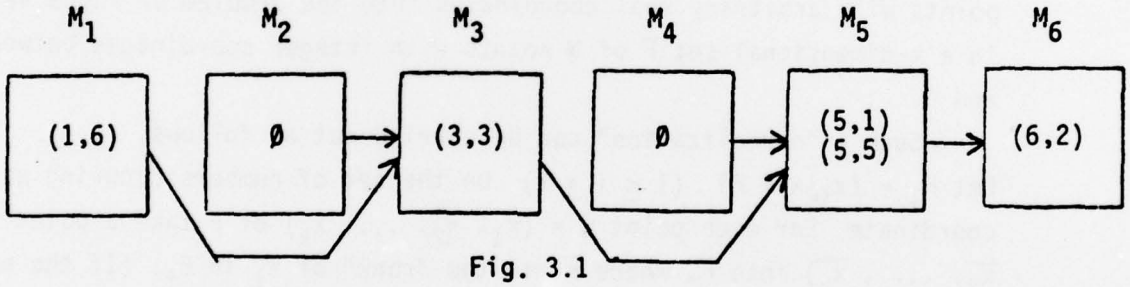


Fig. 3.1

Before discussing the notion of a k-range for k > 1 and how k-ranges are used to store k-dimensional point sets F, one more notation is to be mentioned. For all i,j,t with $1 \leq i \leq j \leq N$ and $1 \leq t \leq k$ let $F_{i,j}^{(t)}$ be that subset of F containing all points whose t-th coordinate is between i and j (both inclusive), i.e.

$$F_{i,j}^{(t)} = \{x \mid x \in F \wedge 1 \leq \overline{x}_t \leq j\}$$

We are now ready to discuss range searches in k dimensions. We first discuss the cases k = 1 and k = 2, and then the general case $k \leq 2$.

In the linear case k = 1 we store F as a 1-range. To process a (normalized) range query (L_1,H_1) we list all elements of $M(L_1)$, $M(L_1 + 1)$, ..., $M(H_1)$. Due to the pointer mechanism employed this takes O(A) time,

where A is the number of points found. To compute the query time we note that roughly 2 lg N steps have to added for normalization.
Taking into account the preprocessing time for normalization of O(N lg N) and of setting up the linear array M we clearly have

$P(N,1) = O(N \lg N)$,

$S(N,1) = O(N)$, and

$Q(N,1) = O(\lg N + A)$.

Consider next the planar case k = 2. We store F as 2-range: the 2-range for F is obtained by storing each of the sets $F^{(2)}_{i,j}$ for $1 \leq i \leq j \leq N$ as 1-range $R_{i,j}$ and by setting up a two dimensional array P of pointers, each element $P_{i,j}$ $(1 \leq i \leq j \leq N)$ pointing to $R_{i,j}$. Thus, to carry out a range search $[L_1, H_1]$, $[L_2, H_2]$ we just have to range search in the 1-range $F^{(2)}_{L_2,H_2}$ for $[L_1, H_1]$.

Example 3.2

Consider the points F as plotted in Fig.3.2

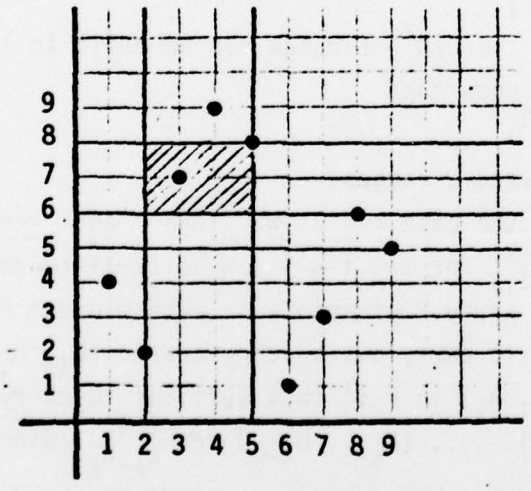


Fig.3.2 Point set F.

To answer the range search [2,5], [6,8] in F we have to range search for [2,5] in $F^{(2)}_{6,8}$ = {(8,6), (3,7), (5,8)} which is available as the 1-range in Fig.3.3

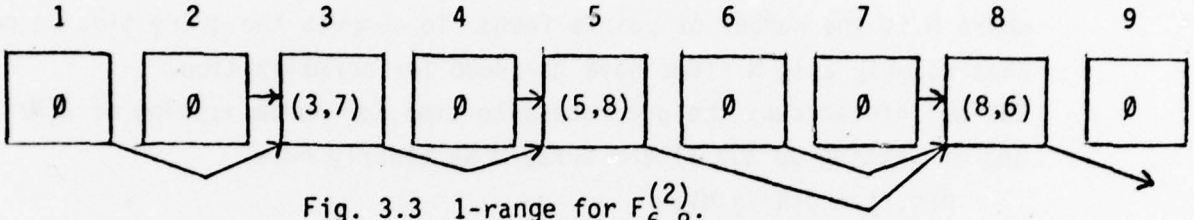


Fig. 3.3 1-range for $F^{(2)}_{6,8}$.

The answer is determined by starting at element #2 in the chain and following the pointers until element #5 is passed: the answer (3,7), (5,8) is obtained as desired.

To analyze the 2-range we note that the cost of performing a query is the sum of three costs: normalization, accessing the 1-range, and searching the 1-range. Since those have costs respectively of 4 lg N, O(1), and O(A), the cost of querying a 2-range is

$$Q(N,2) = O(\lg N + A).$$

The storage required by a 2-range is the sum of the storage required by all 1-ranges. Since there are $\binom{N+1}{2} = O(N^2)$ 1-ranges requiring O(N) storage each, the total storage used is

$$S(N,2) = O(N^3).$$

And since each of the $O(N^2)$ 1-range can be built in linear time after normalization, we know that

$$P(N,2) = O(N^3).$$

We have thus analyzed 2-ranges.

Consider now the case $k \geq 2$. We store F as a <u>k-range</u> as follows. Store first all $F^{(k)}_{i,j}$ for $1 \leq i \leq j \leq N$ as (k-1)-ranges $R_{i,j}$. Now construct a two dimensional array P of pointers, each element $P_{i,j}$ $(1 \leq i \leq j \leq N)$ pointing to $P_{i,j}$. To carry out a range search $[L_1, H_1]$, $[L_2, H_2]$, ..., $[L_{k-1}, H_{k-1}]$, $[L_k, H_k]$ in F it thus suffices to carry out a range search $[L_1, H_1]$, $[L_2, H_2]$, ..., $[L_{k-1}, H_{k-1}]$ in $F^{(k)}_{L_k,H_k}$. Since this is stored as (k-1)-range this process continues until it remains to range search for $[L_1, H_1]$ in a 1-range the latter (as explained above) requiring O(A) steps, A the number of points determined. Since the normalization for each query requires 2k lg N comparisons, the total cost for a query in a k-range is

$$Q(N,k) = O(k \lg N + A)$$

for $k \leq 2$. We will show in Section 4 that this query time is optimal in any "comparison based" model.

We analyze the preprocessing and storage requirements of k-ranges by induction on k, using as the basis for our induction the fact that

$$S(N,2) = P(N,2) = O(N^3).$$

Since storing an N-element k-range involves storing $\binom{N+1}{2}$ N-element (k-1)-ranges, we have the recurrence

$$S(N,k) = O(N^2) \cdot S(N,k-1)$$

which has solution

$$S(N,k) = O(N^{2k-1}).$$

A similar analysis shows that the preprocessing cost of k-ranges is

$$P(N,k) = O(N^{2k-1}).$$

3.2 Multi-level k-ranges

The k-ranges of Section 3.1 provide extremely efficient range searching query time at the expense of high preprocessing and storage costs. In this section we will show how to modify k-ranges to become "ℓ-level k-ranges" which maintain the logarithmic query time while reducing the other costs. We will accomplish this by first developing a set of efficient planar structures and then applying those to successively higher dimensions. Throughout this section we will assume that the points to be searched and the queries have been normalized as in the previous section.

The essential feature of the rapid retrieval times of 2-ranges is that they were based on a covering of all possible y-intervals of interest to range searching. This covering was the "complete" covering, which explicitly stored all y-intervals. Although the complete covering made possible rapid query time, it forced us to store all $O(N^2)$ 1-ranges. We will now investigate other coverings of N intervals which (slightly) increase query time but significantly decrease storage and preprocessing costs.

The first such covering we will investigate is based on a <u>two-level</u> structure (the complete covering is a one-level structure). On the first level we consider one "block" which contains $N^{1/2}$ "units" (assume N is a perfect square) which represent $N^{1/2}$ points each. On the first level of the 2-level 2-range we than store all $\binom{N^{1/2}}{2} = O(N)$ consecutive intervals of

units; that is, we store O(N) 1-ranges. For reasons of space economy we now choose to store 1-ranges as arrays sorted by x-value; this requires space proportional to the number points in the particular range stored, rather than proportional to N. The second level of our covering consists of $N^{1/2}$ blocks each containig $N^{1/2}$ units (which are individual points). Within each block we store all possible intervals of units (points) as 1-ranges. This structure is depicted in Figure 3.4 for the case N = 9. In that figure the bold vertical lines represent block boundaries and the regular vertical lines represent unit boundaries; each horizontal line represents a 1-range structure.

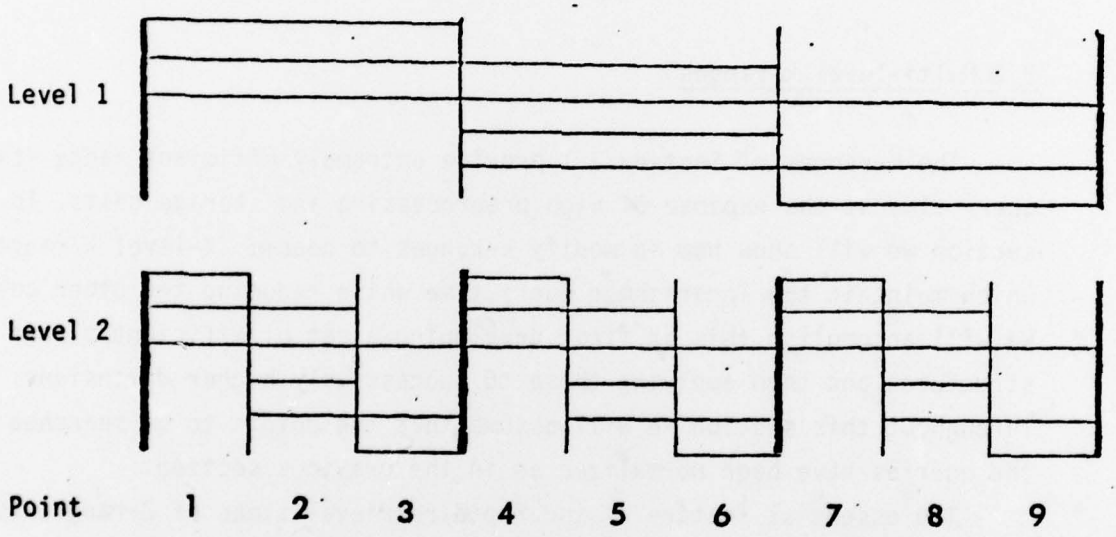


Figure 3.4 A 2-level 2-range.

To answer a range query in a 2-level 2-range we must choose some covering of the particular y-range of the query from the 2-level structure. This can always be accomplished by selecting at most one sequence of units from level one and two sequences of units from level two; this is illustrated in Figure 3.5.

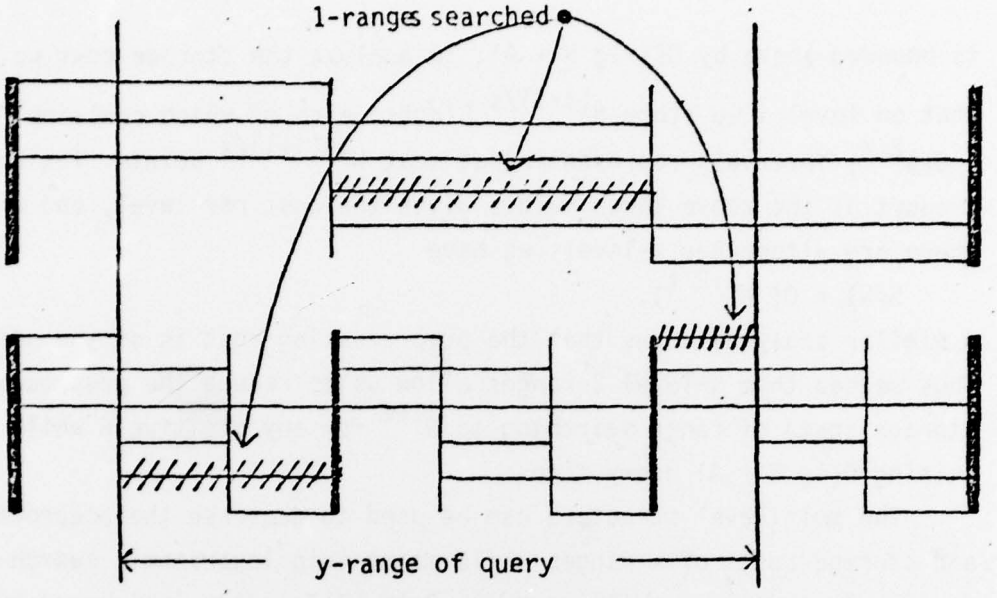


Figure 3.5 Querying a 2-level 2-range.

It is easy to count the cost of a query in a 2-level 2-range: we search a total of at most 3 1-ranges (each at logarithmic cost) and then enumerate the points found, so we have

$$Q(N,2) = O(\lg N + A).$$

To count the storage we note that on the first level we have at most $O(N)$ 1-ranges of size at most N, so that the storage required on the first level is $O(N^2)$. On the second level we have $O(N^{3/2})$ 1-ranges, each representing at most $N^{1/2}$ points, so the storage on that level is also $O(N^2)$. Summing these we achieve

$$S(N,2) = O(N^2).$$

If the points are kept in sorted linked lists as the structures are built, then the obvious preprocessing time of $O(N^2 \lg N)$ can be reduced to

$$P(N,2) = O(N^2).$$

The 2-level 2-range can of course be generalized to an ℓ-level 2-range, a structure consisting of ℓ levels. On the first level there is one block containing $N^{1/\ell}$ units of $N^{1-1/\ell}$ points each. The second level has $N^{1/\ell}$ blocks containing $N^{1/\ell}$ units of $N^{1-2/\ell}$ points, and so on. On each level we store as 1-ranges all $\binom{N^{1/\ell}}{2}$ intervals of units in each block. To answer a query we select an appropriate covering of the query's y-range and then perform searches on those 1-ranges. In such a search we must search at most two intervals on each of the ℓ-levels, so the total cost of the search

is bounded above by $O(\ell \cdot \lg N + A)$. To analyze the storage cost we note that on level i we store $N^{(i-1)/\ell}$ blocks, each of which contains $\binom{N^{1/\ell}}{2}$ $= O(N^{2/\ell})$ intervals representing at most $N^{1-(i+1)/\ell}$ points. Taking the product of the above three values gives the cost per level, and since there are altogether ℓ-levels we have

$$S(N) = O(N^{1+2/\ell}).$$

A similar analysis shows that the preprocessing cost is of the same order. Thus we see that ℓ-level 2-ranges allow us to reduce the preprocessing and storage costs of range searching to $N^{1+\epsilon}$ for any positive ϵ while maintaining $O(\lg N + A)$ query time.

The multilevel structure can be used to decrease the preprocessing and storage costs of k-ranges while maintainig logarithmic search time. To illustrate this we will consider 2-level 3-ranges, which are built by covering z-ranges with 2-level 2-ranges. On the first level of such a structure we have one block of $N^{1/2}$ units representing $N^{1/2}$ points each, and we store all intervals of those units as 2-level 2-ranges. On the second level we have $N^{1/2}$ blocks of $N^{1/2}$ units (which represent one point each), and we store all intervals of units within each block as a 2-level 2-range. Any query can be answered by covering its z-range with one interval from the first level and two intervals from the second level, so we maintain query time of

$$Q(N,3) = O(\lg N + A).$$

We store $O(N)$ 2-level 2-ranges on the first level, and that requires $O(N^3)$ storage. On the second level we store $O(N^{1/2})$ blocks of $O(N)$ 2-level 2-ranges, each of size at most $O(N^{1/2})$ (so those 2-ranges require at most $O(N^{1/2})^2 = O(N)$ storage each). Multiplying these costs we see that the storage required on the second level is $O(N^{5/2})$. Thus the total storage cost is

$$S(N,3) = O(N^3).$$

Using presorting the preprocessing can also be done in cubic time.

The general 2-level k-ranges are inductively built out of 2-level (k-1)-ranges. The k-dimensional structure is built with two levels: on the first there are N (k-1)-dimensional structures of size at most N each and on the second there are $N^{3/2}$ (k-1)-dimensional structures of size at most $N^{1/2}$ each. Since the total storage query cost increases by at most

a factor of three at each dimension, we have for any fixed k

$$Q(N,k) = O(\lg N + A).$$

One can also show that the preprocessing and storage costs grow by a factor of most N for each dimension "added", so we know that

$$P(N,k) = S(N,k) = O(N^k).$$

The above generalization of 2-level k-ranges can also be applied to ℓ-level k-ranges. As we "add" each new dimension we increase the query time by a factor of at most 2ℓ and increase the preprocessing and storage costs by a factor of $O(N^{2/\ell})$. By choosing ℓ as a function of k and ϵ, for any fixed values of k and $\epsilon>0$ we can obtain a structure with performance

$$P(N,k) = S(N,k) = O(N^{1+\epsilon}), \text{ and}$$

$$Q(N,k) = O(\lg N + A).$$

3.3 Nonoverlapping k-ranges

The ℓ-level overlapping k-ranges of Section 3.2 provided logarithmic search time while their preprocessing and space requirements were $O(N^{1+\epsilon})$. In this section we will investigate nonoverlapping ℓ-level k-ranges, which require only O(N lg N) preprocessing and linear space, but have $O(N^{\epsilon})$ query times. We will develop overlapping k-ranges in this section by first presenting and analyzing planar structures, and then investigating the k-dimensional structures.

The first object of our study will be the 2-level nonoverlapping 2-range. On the first level of this structure we consider one block of $N^{1/2}$ units, each unit representing a set of $N^{1/2}$ points contiguous in the y-direction; we then sort the points in each of those units by x-value. The second level of the structure consists of $N^{1/2}$ blocks of $N^{1/2}$ units, each representing a single point. We can represent both levels of the structure by an $N^{1/2}$ by $N^{1/2}$ array: each row of the array represents a "continguous slice" of y-values of the point set and is then sorted by x-value. This structure requires only linear storage and can be built (by $N^{1/2}$ distinct sorts) in O(N lg N) time. Suppose now that we are to do a range search defined by an x-range and a y-range: for all the contiguous y-strips contained wholly in the y-range we can perform two binary searches to give the set of all points contained in the x-range. Since there are

only $N^{1/2}$ such strips altogether and each can be searched in logarithmic time, the total cost of this step is $O(N^{1/2} \lg N + A)$. We can then do a simple scan over the two end y-strips (top and bottom) to see if they contain any points in both x and y ranges; this costs at most $O(N^{1/2})$ to examine the $2N^{1/2}$ points. Thus the total cost of searching is $O(N^{1/2} \lg N)$ and the performance of the structure as a whole is

$P(N,2) = O(N \lg N)$,
$S(N,2) = O(N)$, and
$Q(N,2) = O(N^{1/2} \lg N + A)$.

Nonoverlapping 2-ranges can easily be extended to be multilevel. In the first level of a 3-level 2-range we have one block of $N^{1/3}$ units, each representing $N^{2/3}$ points and sorted by x-value. On the second level we have $N^{1/3}$ blocks, each containing $N^{1/3}$ units of $N^{1/3}$ points contiguous in y (sorted by x). The third level then contains $N^{2/3}$ blocks of $N^{1/3}$ units (points) each. This structure requires storage linear in N and can be built in $O(N \lg N)$ time. To answer a range query we must search at most $N^{1/3}$ units on the first level and $2N^{1/3}$ units on each of the second and third levels. The cost of each of those searches is logarithmic (excluding the manipulation of points found), so the total cost of searching is $O(N^{1/3} \lg N)$. The obvious extension to ℓ-level nonoverlapping 2-ranges carries through without flaw and has performance

$P(N,2) = O(\ell N \lg N) = O(N \lg N)$,
$S(N,2) = O(\ell N) = O(N)$, and
$Q(N,2) = O(N^{1/\ell} \lg N + A)$.

Note that for any fixed $\epsilon > 0$ we can choose $\ell > 1/\epsilon$ and achieve a structure with linear storage, $O(N \lg N)$ preprocessing, and $O(N^{\epsilon})$ search time.

Nonoverlapping ℓ-level 2-ranges can be generalized to nonoverlapping ℓ-level k-ranges for each dimension we "add" we use the same multilevel structure and store the units as ℓ-level nonoverlapping (k-1)-ranges. As each dimension is added the storage remains linear and the preprocessing remains $O(N \lg N)$ (with increased constants). The search time, however, increases by a factor of $N^{1/\ell}$ for each added dimision. Thus by choosing ℓ as a function of k and ϵ one can acieve performances

$P(N,k) = O(N \lg N)$,
$S(N,k) = O(N)$, and
$Q(N,k) = O(N^{\epsilon} + A)$.

Note that this is for fixed ϵ, k and ℓ: if ℓ is allowed to vary with N then one achieves a tree-like structure (specifically, the range trees of Bentley [2] if $\ell = \lg N$).

4. Lower bounds

We have shown in Section 3 by using k-ranges that a k-dimensional set of N points can be stored such that range queries can be answered in time $O(k \lg N + A)$. We will now demonstrate that this is optimal.

For an arbitrary point set F, let R(F) be the number of different range queries possible for F. (We say that two range queries are different iff their answers are different). Let

$R(N,k) = \max \{R(F) \mid F \text{ a set of N points in k dimensions}\}$.

It is easy to see that $R(N,1) = \binom{N+1}{2} + 1$, for the answer to a range query is either enpty (1 such answer) or can be defined by two of N+1 interpoint locations.

The exact value of R(N,k) for general N and k seems more difficult to calculate. We can immediately observe that $R(N,k) \leq \binom{N+1}{2}^k$, since for each dimension there are only N + 1 essentially different positions for upper and lower bounds for a range search. One might wonder how close R(N,k) can grow to this bound; Theorem 4.1 partially answers this.

Theorem 4.1

$$R(N,k) \geq \left(\frac{N}{2k}\right)^{2k}.$$

Proof: To avoid complications assume N is a multiple of 2k. Let F be the set of all points with a single nonzero integer coordinate in the closed interval $[\frac{-N}{2k}, \frac{N}{2k}]$. Consider the set of all range queries $[L_1, H_1]$, $[L_2, H_2]$, ..., $[L_k, H_k]$ with $-\frac{N}{2k} \leq L_i \leq -1$ and $1 \leq H_i \leq \frac{N}{2k}$ for i = 1,2, ..., k. The $\left(\frac{N}{2k}\right)^{2k}$ range queries obtained in this way clearly determine different sets of points, and the result follows. □

Corollary

For range queries on N points in k dimensions O(k lg N + A) is optimal for "comparison-based" methods.

Proof

By the Theorem, $R(N,k) \geq (\frac{N}{2k})^{2k}$. Hence any algorithm for range queries based on binary decisions requires in the worst case at least $\lg (\frac{N}{2k})^{2k} =$ $= 2k \lg N - 2k \lg k - 2k \lg 2$ steps. Hence the 2k lg N comparisons used for range searching in 1-level k-ranges is optimal to within second-order terms. □

5. Conclusions

We have presented three variants of a new data structure (the k-range) for storing k-dimensional sets of N points and permitting fast responses to range queries. The first variant, one-level k-ranges, requires only 2k lg N comparisons per query, plus an amount of list processing proportional to A, the number of answers found. However, prepocessing and storage costs of $O(N^{2k-1})$ are prohibitively high. With the second variant, multi-level k-ranges, lookup time is still O(lg N + A), but preprocessing and storage costs are reducable to $O(N^{1+\epsilon})$ for every fixed $\epsilon > 0$. Employing the third variant, nonoverlapping k-ranges, storage can be reduced to O(kN), preprocessing to O(N lg N) and for every $\epsilon > 0$ a worst case query time of $O(N^{\epsilon} + A)$ can be achieved.

The results are summarized and compared with previously known techniques in the following table (Fig. 5.1), showing the behaviour for fixed k and large N.

Structure	P(N,k)	S(N,k)	Q(N,k)
Naive	$O(N)$	$O(N)$	$O(N)$
k-d Trees	$O(N \lg N)$	$O(N)$	$O(N^{1-1/k} + A)$
Nonoverlapping k-ranges	$O(N \lg N)$	$O(N)$	$O(N^{\epsilon} + A)$
Range Trees	$O(N \lg^{k-1} N)$	$O(N \lg^{k-1} N)$	$O(\lg^{k} N + A)$
(Overlapping) ℓ-level k-ranges	$O(N^{1+\epsilon})$	$O(N^{1+\epsilon})$	$O(\lg N + A)$
(Overlapping one level) k-ranges	$O(N^{2k-1})$	$O(N^{2k-1})$	$O(\lg N + A)$

Fig. 5.1

Fast solutions to other problems involving point sets in k-dimension can also be obtained by using the data structures and techniques of this paper. Two such examples are the problem of computing the Emperical Cumulative Distribution Function (ECDF searching problem) and the Maxima searching problem discussed in detail in Bentley [2]. For a point x, the ECDF searching problem and the maxima searching problem can be formulated as follows:

ECDF searching problem: Determine the number of points y in F with $y_j \leq x_i$ for i = 1,2, ..., k.

Maxima searching problem: Determine if there exists a point y in F with $y_i > x_i$ for i = 1,2, ..., k.

It is easy to see that by formulating the above problems in terms of range searching the table in Fig. 5.1 is also valid for the ECDF searching problem and the maxima searching problem. (Indeed, the contribution of A can be ignored. This is evident for the maxima searching problem since the answer is only "yes" or "no". For the ECDF searching problem it follows from the fact that A, as the count of the number of points determined, can be obtained in O(lg N) rather than O(A) time, by storing the 1-ranges involved as sorted arrays.)

Despite the further insights into range searching gained by this paper, a number of open problems remain. Are there other data structures

with a still better tradeoff between P(N,k), S(N,k), Q(N,k)? In particular, are a total of 2k lg N comparisons is $O(N^{2k-1})$ optimal for space and storage? Can the product $P(N,k) \cdot S(N,k) \cdot Q(N,k)$ be reduced to $O(N^2 \lg^2 N)$? If not, can one show lower bounds on the above product, indicatin "space-time" tradeoffs. What is the situation when the dynamic case (insertion and deletion of points in between queries) is considered?

Another problem of independent interest is the exact computation of R(N,k) of section 4. Although we have shown $(\frac{N}{2k})^{2k} \leq R(N,k) \leq \binom{N+1}{2}^k$ we have been unable to compute the exact value of R(N,k) in general. We do not even know the values of R(N,2) except for small N.

In this paper we have presented (asymptotically) fast worst case methods for range searching, some of them with (asymptotically) small amount of preprocessing and storage. We do not nesessarily advocate these methods for practical applications. The results do, however, suggest that it may be possible to find methods for solving range queries efficiently both as far as average and worst-case behaviour are concerned.

References

1. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Comm. ACM 18, 9. pp. 509-517, 1975

2. Bentley, J.L.: Multidimensional Divide-and-Conquer. To appear in Carnegie Mellon University Computer Science Department Research Review, 1978.

3. Bentley, J.L., Friedman, J.H.: Algorithms and data structures for range searching. Submitted for publication.

4. Knuth, D.E.: The Art of Computer Programming, vol. 3. Addison-Wesley, Reading, Mass. 1973.

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

6. Maurer, H.A., Ottman,).: Manipulating sets of points - a survey. In: Graphen, Algorithmen, Datenstrukturen: Workshop 78, Hauser, München-Wien (1978).

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-78-136	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) TWO PAPERS ON RANGE SEARCHING		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) J.L. Bentley, J.H. Friedman, and H.A. Maurer		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Schenley Park Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE August 1978 13. NUMBER OF PAGES 45
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same as above		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report contains two independent papers on range searching. A range search retrieves from a file all records which conjunctively satisfy a set of range requirements for the keys; that is, each key must lie in some specified range. Range searching arises in many applications, such as data base management and statistical computing. The first paper in this report, "A survey of algorithms and data structures for range searching" by J. L. Bentley and J. H. Friedman, describes the known "logical

CONT'

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

cont'

structures" which can be used for range searching and then discusses the implementation of those structures in different storage media. This paper is slanted towards the practitioner. The second paper, "Efficient worst-case data structures for range searching" by J. L. Bentley and H. A. Maurer, is more theoretical. Two new classes of data structures are proposed for range searching, establishing bounds on the asymptotic complexity of the problem.