Chapter 41

The P-range Tree: A New Data Structure for Range Searching in Secondary Memory

Sairam Subramanian[†] **Sridhar Ramaswamy**[‡]

Abstract

External 2-dimensional range searching is a fundamental problem with many applications in relational, object-oriented, spatial, and temporal databases. For example, interval intersection can be reduced to 2-sided, 2-dimensional searching and indexing class hierarchies of objects to 3-sided, 2-dimensional searching. In this paper we introduce a new data structure called the *p-range tree* for performing 2-dimensional range searching in secondary memory. Given n points on the plane and a disk with block size B, the p-range tree uses $O(\frac{n}{B})$ disk blocks of storage and answers 2- and 3-sided queries in $O(\log_B n + t/B + \mathcal{IL}^*(B))$ disk I/O's.[1] Here t is the number of points in the query. We also give a secondary memory data structure for answering general 2-dimensional queries with the same I/O bound.

Using P-range trees we also give data structures for range searching in main memory. For orthogonal range searching in d-dimension the data structure occupies $O(n \log^{d-1} n)$ space and answers queries in $O(\mathcal{IL}^*(n) \log^{d-2} n + t)$ time. The time-space product of this data structure is better than that of previously known data structures [5,25].

We also give a lower bound for range-searching in secondary memory that shows that 2-dimensional range searching is inherently harder in secondary storage than in main memory. We show that in order to be able to answer any 2-dimensional query in $O(\log_B^c n + t/B)$ disk I/O's (where c is an arbitrary positive constant), a data structure must occupy $\Omega(\frac{n}{B}(\log \frac{n}{B})/\log\log_B n)$ disk blocks.

*This research was conducted while the authors were at Brown University. Supported by NSF PYI award CCR-9157620, Honeywell, Thinking Machines, and Xerox Corporations, and ARPA contract N00014- 91-J-4052 ARPA Order No. 8225

[†]University of Texas at Austin. Email: sai@cs.utexas.edu

[‡]Bell Communication Research. Email: sr.cs.brown.edu

[1] $\mathcal{IL}^*$ denotes the *iterated* log* function. $\mathcal{IL}^*(n)$ is the number of times we must repeatedly apply the log* function to n before the result becomes ≤ 2.

1 Introduction and motivation

The successful realization of any data model in a large-scale database requires supporting its language features with efficient secondary storage manipulation. Consider the relational data model of [8]. While the declarative programming features (relational calculus and algebra) of the model are important, it is crucial to support these features with data structures for searching and updating that make optimal use of secondary storage. B-trees and their variants B$^+$-trees [1,9] are examples of such data structures. They have been an unqualified success in supporting external dynamic 1-dimensional range searching in relational database systems.

The general data structure problem underlying efficient secondary storage manipulation for many data models is external d-dimensional range searching. In particular, 2-dimensional range searching (in various forms) is crucial in the implementation of object-oriented, hierarchical, temporal, spatial, and constraint databases [18,19,27,35]. B-trees, that perform 1-dimensional range searching well, are inefficient at handling more general problems like two and higher dimensional range searching.

The problem of 2-dimensional range searching in both main memory and secondary memory has been the subject of much research. Many elegant data structures like the range tree [3], priority search tree [22], segment tree [2], and interval tree [12,13] have been proposed for use in main memory for 2-dimensional range searching and its special cases (see [7] for a detailed survey). Most of these algorithms are not efficient when mapped to secondary storage. However, the practical need for good I/O support has led to the development of a large number of empirical external data structures [15,16,21, 23,24,28,29,30,32] which do not have good theoretical worst-case bounds but have good average-case behavior for common spatial database problems. The worst-case performance of these data structures is much worse than the optimal bounds achievable for dynamic external 1-dimensional range searching using B$^+$-trees. (See [19] for a more complete reference on the field.)

In this paper, we study the problem of 2-

dimensional range searching in secondary memory. We present a secondary storage data structure for this problem that is space-optimal and answers queries in time that is within a very small additive term of the optimum. We also present secondary memory data structures for important special cases of 2-dimensional range searching. These data structures occupy optimal linear storage and answer queries in nearly optimal time. Our data structures also allow updates to be made in amortized logarithmic time. These data structures are based upon a new 2-dimensional searching structure called the *p-range tree* that we show has applications for range searching in main memory as well. Using p-range trees, we give efficient static data structures for d-dimensional range searching in main memory. The space-time product of our main memory data structure is better than any previously known data structure for range searching in dimension ≥ 3.

We also prove a lower bound on the number of disk blocks needed for constructing an efficient general 2-dimensional range searching data structure in secondary memory. Our lower bound shows that it is not possible to efficiently map a main memory data structure for general 2-dimensional range-searching on to secondary memory without incurring a space overhead.

Preliminaries: We now introduce our model for secondary storage algorithms. We make the standard assumption that each secondary memory access transmits one disk block or B units of data, and we count this as one I/O.

Function symbols and conventions: The symbol $\mathcal{IL}^*$ will denote the *iterated* $\log^*$ *function*. $\mathcal{IL}^*(n)$ is the number of times we must apply the $\log^*$ function to n before the result becomes ≤ 2. It is also the 3rd row inverse of the Ackerman function. $\mathcal{IL}^*(n)$ is the inverse of $A(3,n)$ [26] where A denotes the Ackerman function. Unless otherwise stated all logarithms will be with respect to base 2. Also, throughout this article we will use the terms in-core and main memory to mean the same thing.

The efficiency of our algorithms will be measured in terms of the number of I/O operations they perform. For example, consider the problem of 1-dimensional range searching. Given a set P of points on the real line, a 1-dimensional query q is of the form $a \leq x \leq b$ and the answer to q consists of all the points $x \in P$ such that $a \leq x \leq b$. B-trees can be used to answer this query in $O(\log_B n + t/B)$ disk I/O's in the worst case which is optimal. Here, t denotes the number of points in P that belong to q. Insertions and deletions of points can also be accomplished with $O(\log_B n)$ I/O's.

Two important special cases of 2-dimensional range searching are 2-sided and 3-sided range searching (see Figure 1). 2-sided range searching can be used to index constraint databases [19,27], and 3-sided range searching to index object-oriented databases [20,35]. The priority search tree can answer 3-sided queries (and therefore, 2-sided queries as well) in-core in time $O(\log n + t)$, using storage $O(n)$ with a worst-case update time of $O(\log n)$.

Previous attempts to answer 2 and 3-sided queries in external memory by implementing priority search trees in secondary memory [17,19] did not have optimal query times. [17] uses optimal storage but answers queries in $O(\log n + t/B)$ time. [19] improves on this, answering queries in $O(\log_B n + \log B + t/B)$ time using optimal storage. Neither of them allow inserts and deletes from the data structure. In [27] we presented a data structure to solve this problem using a technique called *path caching* that answers queries in optimal time $O(\log_B n + t/B)$, but uses sub-optimal storage.

In this paper, we present a data structure, the p-range tree, that can answer 3-sided range queries in $O(\log_B n + t/B + \mathcal{IL}^*(B))$ I/O's. Our data structure occupies $O(n/B)$ disk blocks of storage. Points can be inserted/deleted in $O(\log_B n + (\log_B n)^2/B)$ disk I/O's. Using p-range trees, we present a data structure for answering general 2-dimensional queries in $O(\log_B n + t/B + \mathcal{IL}^*(B))$ I/O's that occupies $O((n/B)\log n/\log\log_B n)$ disk blocks. P-range trees have applications in main memory as well. Given a set of n points in d-dimensional space ($d \geq 3$), we use p-range trees to devise a main memory data structure to answer range queries in $O(t + \mathcal{IL}^*(n)\log^{d-2} n)$ time using $O(n\log^{d-1} n)$ storage.

In main memory, Chazelle [6] has given a strong lower bound for the problem of (static) range searching in 2 dimensions. He shows that on a pointer machine a query time of $O(t + \log^c n)$, where t is the number of points to be reported and c is a constant, can only be achieved at the expense of $\Omega(n(\log n/\log\log n))$ storage. In Section 5 we show that on an *external memory pointer machine*, a query time of $O(t/B + \log_B^c n)$, where t is the number of points to be reported and c is a constant, can only be achieved at the expense of $\Omega((n/B)\log n/\log\log_B n)$ disk blocks of storage. The external memory pointer machine is a natural generalization of the pointer machine model [34] and is suitable for analyzing secondary memory algorithms. Our lower bound shows that we cannot implement an efficient data structure for 2-dimensional range searching in external memory using $O(n\log n/B\log\log n)$ disk blocks of storage. Thus in order to perform 2-dimensional range searching in secondary memory we must use more space than the main memory data structure. For instance consider the normal situation where $B = \Omega(n^\epsilon)$ for some

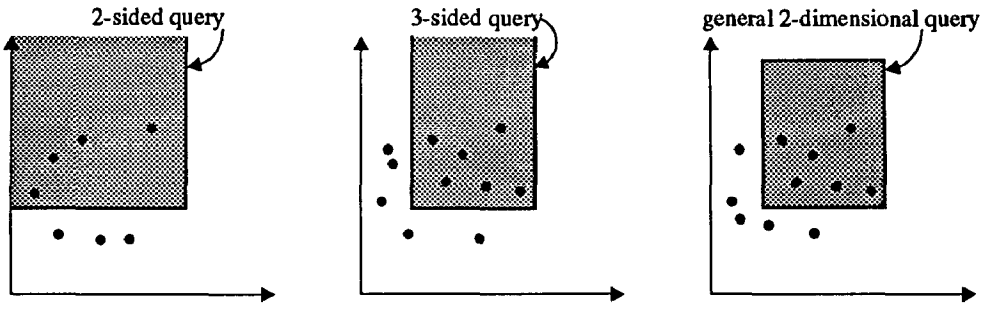

Figure 1: 2-sided, 3-sided and general 2-dimensional range queries.

constant $\epsilon < 1$. In this scenario any secondary memory data structure must use $\Omega(\frac{n}{B}\log n/)$ disk blocks of storage while the main memory data structure requires only $O(n\log n/\log\log n)$ storage.

The rest of the paper is organized as follows. Section 2 presents the basic p-range tree which can be used to answer 2-sided queries in $O(\log_B n + \mathcal{IL}^*(B) + t/B)$ I/O's using optimal linear storage. Section 3 extends the p-range tree to answer 3-sided queries with the same bounds. Section 4 presents a data structure to answer general 2-dimensional queries in $O(\log_B n + \mathcal{IL}^*(B) + t/B)$ I/O's using $O((n/B)\log n/\log\log_B n)$ disk blocks of storage. It also shows how p-range trees can be used in main memory. Section 5 presents the lower bound for 2-dimensional range searching in secondary memory. We finish with the conclusions and open problems in Section 6.

2 The basic p-range tree: A data structure for 2-sided queries

In this section we describe the basic structure of the p-range tree and show how to use it to answer 2-sided queries. The data structure uses optimal $O(\frac{n}{B})$ disk blocks of storage and answers any 2-sided query in $O(\log_B n + t/B + \log^*(\frac{n}{B}))$ I/O's, where t is the number of points that belong in the query.

In Section 3 we show how to answer 3-sided queries using the p-range tree. We then describe a more sophisticated version of the data structure that improves the query performance to $O(\log_B n + t/B + \mathcal{IL}^*(B))$ I/O's per query. In Section 4 we address the problem of answering general 2-dimensional queries.

2.1 The basic p-range tree: The p-range tree is a combination of a priority search tree and a range tree (in two dimensions). Given the set P of n points on the plane we construct a p-range tree as follows.

We first identify the top B y-points in P (points that are maximum in terms of their y-values) and store them at the root r. We then repeatedly divide the remaining points into equal halves on the basis of their x-values until we get $O(\frac{n}{B}/\log^2(\frac{n}{B}))$ sets s_1 through s_k each with no more than $B\log^2(\frac{n}{B})$ points. This division is recorded in the natural form as a binary search tree T_r rooted at r. The root node is called a *layer 0* node, while the nodes $n_1, n_2, \cdots, n_k$ corresponding to the sets s_1 through s_k are called *layer 1* nodes. The internal nodes of T_r are called *non-layer nodes.*

For every set s_i we copy the top B y-points in s_i on every node along the n_i-to-root path. At each node z in the tree T_r we sort the points copied at node z by their y-values in decreasing order and store the sorted list in blocks of size B. This list is called the *descendant list* (or the D-list) of node z. For any node n_i in layer 1 let $v_1, v_2, \cdots v_q$ be the right siblings (if they exist) of the nodes along the n_i-to-root path. At node n_i we keep a copy of the top B y-points from each of the right siblings $v_1, v_2, \cdots v_q$. These points are sorted according to their y values in decreasing order and stored in a blocked fashion in a list at node n_i. This list is called the *ancestor list* (or the A-list) of n_i.

Let β denote an arbitrary layer 1 node n_i. To build the p-range tree we continue the division recursively. We start by storing the top B points (by their y-value) (of the set associated with β) at β. We then divide the remaining points repeatedly (on the basis of their x values) until we get $O(B\log^2(\frac{n}{B})/(B(\log\log(\frac{n}{B}))^2)) = O(\log^2(\frac{n}{B})/(\log\log(\frac{n}{B}))^2)$ sets of size no more than $B(\log\log(\frac{n}{B}))^2)$ each. As before, this division is recorded in the form of a binary search tree T_β rooted at β. For every final set s corrsponding to a leaf of T_β we also copy the top B points of s in all the nodes along the path in T_β from s to β. The copied points at the nodes of T_β are sorted in decreasing order by their y-values and stored in D-lists as before.

The leaves of T_β (for all β) corresponding to the sets generated in this second phase are called *layer 2* nodes, while the other nodes in the tree T_β are called non-layer nodes. As before, we create A-lists at every node α in layer 2 by copying the top B y-points from

the right siblings along the path from α to its ancestor n_i in layer 1.

We continue in this fashion until we reach sets that have only B nodes each. For every node α in layer i we copy-up the top B y-points in α at the nodes along the α-to-β path (where β is the $i-$ 1rst layer-node that is the ancestor of α). At α we also copy the top B y-points from the right siblings along the α-to-β path. As before, we construct the corresponding A- and D-lists. Figure 2 gives a pictorial representation of the p-range tree T constructed in this fashion. We note that the points in P are stored in the layer-nodes at various layers while the non-layer nodes only contain copies in their D-lists.

Consider the p-range tree constructed as described above. If we imagine each layer of nodes to be the direct children of their corresponding ancestors in the previous layer and ignore all the non-layer nodes then the tree T is just like a priority search tree (except for the A-lists) of depth $O(\log^*(\frac{n}{B}))$. In other words if we compress all the non-layer nodes then we get a priority search tree. If we look at the portion of the tree between any two layers then the various subtrees look like range trees. Since they are search trees in the x-dimension and points (albeit only some) at the leaves get copied along the root-to-leaf path.

LEMMA 2.1. *The p-range tree T constructed as described has depth $O(\log(\frac{n}{B}))$ and has $O(\log^*(\frac{n}{B}))$ layers. Furthermore, the number of disk blocks required to store the A- and D-lists at all the nodes in T is $O(\frac{n}{B})$.*

Proof. Since T is a search tree on the x-values and the bottom most nodes have $O(B)$ points each, it follows that the depth of the tree is $O(\log(\frac{n}{B}))$. The set at layer 0 (the root) contains n points. The sets at layer 1 contain $B\log^2(\frac{n}{B})$ points each. The sets at layer 2 contain $B(\log\log(\frac{n}{B}))^2$ points each, and so on. Thus, if sets at the ith layer contain $B(\log z)^2$ points then the sets at the $i+$ 1rst layer contain $B(\log\log z)^2$ points. Therefore after $O(\log^*(\frac{n}{B}))$ layers each set contains at most B points.

The number of nodes in layer 1 is $O(\frac{n}{B}/\log^2(\frac{n}{B}))$. From the sets at each of these nodes n_i we copy B points to nodes along the n_i-to-root path in T_r. Each layer 1 node also has an A-list that contains B points from the right siblings of each of its ancestors. The height of T_r is easily seen to be $O(\log(\frac{n}{B}))$. Therefore, each layer 1 node contributes $O(B\log(\frac{n}{B}))$ copies to the D-lists of its ancestors. Also, the size of the A-list at any layer 1 node is $O(B\log(\frac{n}{B}))$. Since there are $O(\frac{n}{B}/\log^2(\frac{n}{B}))$ nodes in layer 1 the total number of points in the A-lists at layer 1 and in the D-lists of their ancestors is $O(\frac{n}{B}/\log^2(\frac{n}{B})) \times O(B\log(\frac{n}{B})) = O(n/\log(\frac{n}{B}))$. Therefore, we need $O(\frac{n}{B}/\log(\frac{n}{B}))$ disk blocks to store these points. Continuing this we get a geometric series that sums up to $O(\frac{n}{B})$ thus giving us a total space requirement of $O(\frac{n}{B})$ disk blocks.

2.2 Answering queries using the p-range tree: Consider a 2-sided query $q = \{x \geq a, y \geq b\}$. We first search through the tree T and locate the layer-node c that contains the corner point (a, b) of the query. This can be done with $O(\log_B n)$ I/O's by maintaining a skeleton of T (without the A and D-lists) in a blocked fashion. Let π_c be the path from c to the root r of the tree, and let $\alpha_1, \alpha_2, \cdots \alpha_k$ be the $O(\log^*(\frac{n}{B}))$ layer-nodes on π_c. To find the points that belong in the query let us view T as a priority search tree Tp by compressing all the non-layer nodes. In this compressed form we only have layer-nodes and the parent of a node α from layer i is its ancestor β (in T) from layer $i-1$. We then have the following lemma.

LEMMA 2.2. *Every point that belongs in the query is in a layer-node that is either on or to the right of π_c. Furthermore, for any node α in layer i; points stored at α can belong to the query only if:* (a) *Either all the points in its parent β belong to the query, or* (b) *β is one of the nodes $\alpha_1, \alpha_2, \cdots \alpha_k$ on the path π_c.*

Proof. The first claim follows because Tp is a search tree in x-dimension, therefore all the points to the left of π_c have x-values less than a and thus cannot be in the query. To prove the second claim let us suppose some point in α belongs to the query. By the first claim α is either on or to the right of π_c, thus its parent β is also on or to the right of π_c. Suppose β is to the right of π_c. Then, the x-values of all the points stored at β are greater than or equal to a. Furthermore, by construction the y-value of every point stored at β is greater than the y-value of all the points stored at α. Therefore any point at α can belong to the query only if all the points stored at β belong to the query.

We will use Lemma 2.2 to show that we can find the points in the query efficiently. To find the points on the layer-nodes along the path π_c we examine the $O(\log^*(\frac{n}{B}))$ layer-nodes α_1 through α_k on it, and output the points that belong to the query. This requires $O(\log^*(\frac{n}{B}))$ I/O's. We now have to find points that are stored at layer-nodes α to the right of π_c. To find these points we only need to check whether their y-values are greater than or equal to b. This is because by virtue of being on the right of π_c their x-values must be at least a.

2.3 Using A- and D-lists to find points: Consider the nodes at layer i (to the right of π_c) whose parent is the layer $i-1$ node α_{i-1} on π_c. To find the points in these nodes we proceed as follows: Let π_i be the subpath of π_c in T between the ith and $i-$ 1rst layer-nodes α_i

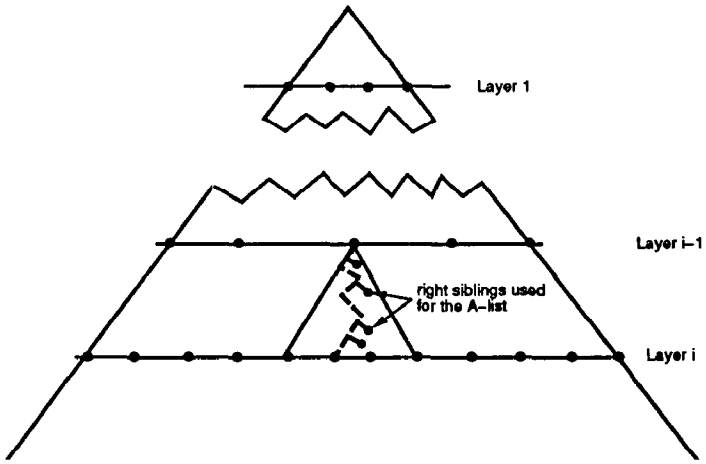

Figure 2: The p-range tree. The leaves contain $O(B)$ points. The number of layers is $O(\log^*(n/B))$.

and α_{i-1}, and let $v_1, v_2, v_3, \cdots, v_l$ be the right siblings along π_i. All the layer i children of α_{i-i} (in Tp) are its descendants in T. Therefore, every layer i child of α_{i-1} (in T_p) that is to the right of π_c is a descendant (in T) of some right sibling v_j. Recall that the points stored in layer i nodes are copied upwards along the tree path in T. Therefore to find the points on these layer i nodes we only need to look at the D-lists of all the right-siblings v_1 through v_l. Furthermore, since the D-list is sorted in decreasing order by the y-value of the points we only need to examine each D-list block-by-block until we find a point with y-value less than b. Thus all but the last I/O in each D-list will give B points that are in the query.

Unfortunately there can be many such right siblings. For example, to find points in layer 1 we may have to examine $O(\log(\frac{n}{B}))$ right siblings. We therefore cannot examine all of them blindly since none of them may yield any points. To see which of the right siblings to examine we make use of the A-list at α_i. Recall that the A-list at α_i stores the top B y-points from the D-lists of the right siblings v_1 through v_l sorted in decreasing order according to the y-values. Therefore, instead of examining all the D-lists we examine the A-list at α_i. The A-list is examined block-by-block until we find a block with points that are not in the query. Thus all but the last I/O performed while examining the A-list are useful. After examining the A-list we examine the D-lists on the right siblings v_1 through v_l. However, we examine the D-list of the sibling v_j if and only if all of the top B y-points from v_j that were copied onto the A-list of α_i were found to be in the query. Thus we examine a D-list only if it has already yielded a block of B points that are in the query.

2.4 Collecting points from the descendants of right siblings: We now only have to find points in layer-nodes whose parents (in T_p) are also to the right of π_c. By Lemma 2.2 for any such node α all the points in its parent β (in T_p) belong to the query. To find these points we therefore start at the top and work our way downwards.

Consider the nodes in layer 1. Since all of these nodes are the children of the root node r (in T_p) that is on π_c they are already taken care of (by the previous mechanism). Let β be one such layer 1 node such that all the points in β were found to be in the query. Now we need to examine the layer 2 children of β (in T_p). Since all of these nodes are to the right of π_c we only have to check and see if the y-values of the points in these nodes are $\geq b$. We follow the same strategy as before. Let α be the leftmost child of β in T_p, and let π_β be the path from α to β in T. As discussed above, the points in the layer i children of β can be found in the D-lists of the right siblings along π_β. We retrieve these points efficiently in the same manner as before by examining the A-list at α and then examining the relevant D-lists as required. Note that we examine the A-list of α only if all the points in β are found to be in the query. In this manner we can retrieve all the points that belong to the 2-sided query efficiently. Apart from the layer-nodes on π_c and their A-lists we examine other A- and D-lists only as needed. Therefore, the total number of I/O's required to retrieve all the t points in the query is $O(\log^*(\frac{n}{B}) + t/B)$. Factoring in the number of I/O's required to find the corner we get our bounds.

3 Using p-range trees to answer three sided queries

In this section we show how to extend the basic p-range tree of from Section 2 to answer 3-sided queries. We also briefly discuss how to improve the query performance of the basic data structure.

We first describe how to extend the p-range tree data structure to answer 3-sided queries in $O(\log_B n + t/B + \log^*(\frac{n}{B}))$ I/O's. We then briefly discuss how to refine the data structure so that 2- and 3-sided queries can be answered in $O(\log_B n + t/B + \mathcal{IL}^*(\frac{n}{B}))$ I/O's.

It is possible to reduce the extra additive term for answering queries from $\mathcal{IL}^*(\frac{n}{B})$ to $\mathcal{IL}^*(B)$ by using a B-tree like structure called the *meta-block tree*. The meta-block tree which was introduced by Kanellakis, Ramaswami, Vengroff, and Vitter [19] allows us to divide the input into meta-blocks of size B^2 such that we need to use the p-range tree only to answer queries within a meta-block. This reduces the additive term to $\mathcal{IL}^*(B)$. Using the metablock tree we can also dynamize our range searching data structures to get good amortized update times. Details can be found in [33]. In particular we get the following bounds.

THEOREM 3.1. *Given a set P of n points on the plane there exists a data structure for answering 2- and 3-sided queries on P that uses $O(\frac{n}{B})$ disk blocks of space. Queries can be answered using $O(\log_B n + t/B + \mathcal{IL}^*(B))$ I/O's where t denotes the number of points of P that belong to the query. Updates require $O(\log_B n + (\log_B n)^2/B)$ amortized disk I/O's per insertion/deletion.*

3.1 P-range tree and 3-sided queries: Consider the compressed version Tp of T. Given a 3-sided query $q = \{a \leq x \leq b, d \leq y\}$ let $c1$ and $c2$ be the nodes of Tp that contain the two corners (a, d) and (b, d) of the query q. Let π_{c1} and π_{c2} be the paths from $c1$ and $c2$ respectively to the root r of the tree T. Then, the points that belong to q are either on one of the paths π_{c1}, π_{c2} or they are in the layer-nodes that are between π_{c1} and π_{c2} in T (layer-nodes that are to the right of π_{c1} and to the left of π_{c2}). Since the query is bounded in the x-dimension on both sides, in order to examine the layer-nodes in between π_{c1} and π_{c2}, we construct A-lists of two kinds. For a node α in layer i we construct the lists Al and Ar. The list Ar contains the top B y-points from the right siblings along the α-to-β path where β is the layer $i-1$ ancestor of α. The list Al contains the top B y-points from the left siblings along the same path.

We can use the two A-lists and the D-lists in a manner similar to the previous case to efficiently find the points in q. The Ar lists are used while proceeding from the nodes on π_{c1} and the Al lists are used while proceeding from the nodes on π_{c2}. This gives us an efficient mechanism for finding the points in our query except where the paths π_{c1} and π_{c2} meet.

3.2 Finding points where π_{c1} and π_{c2} meet: Let z be the node of T where π_{c1} and π_{c2} meet. The node z is called the *top node* of the query q. Suppose that z is between layers $i-1$ and i of T. Let α_1 and α_2 be the layer i nodes on π_{c1} and π_{c2} respectively, and let β be the common layer $i-1$ node on both the paths. To find the points in our query we should only examine the layer i nodes that are descendants of z because the x-values of points in other layer i nodes are either less than a or are greater than b.

Let $u_1, u_2, \cdots, u_k$ and $v_1, v_2, \cdots, v_k$ be the right and the left siblings between layers $i-1$ and i along π_{c1} and π_{c2} respectively. To find the layer i points in our query we should examine the D-lists of only those u_i and v_i that are descendants of z. We are now in trouble because the Al-list at α_1 (and the Ar list at α_2) contain the top B y-points from all the right siblings u_1 through u_k (all the left siblings v_1 through v_k). Therefore these A-lists would give us points that are not in the query. Thus to efficiently search the layer i descendants of z we need A-lists that sample points only from the right and the left siblings that are descendants of z. Since we have no a priori knowledge of where the top node z is going to be, we need to create many Al- and Ar- lists at α_1 and α_2 respectively; one for each possible z. If we have these A-lists we can look at the appropriate Al-list (Ar-list) to efficiently examine the layer i descendants of z.

3.3 Building a p-range tree for 3-sided queries: To build the modified p-range tree we proceed as follows. As before, we keep the top B y-points at the root r and divide the remaining points repeatedly in terms of their x-values and record this division in the search tree T_r. However, now we stop and form layer 1 when we reach sets of size $O(B\log^3(\frac{n}{B}))$ instead of continuing on till we get sets of size $O(B\log^2(\frac{n}{B}))$. We build the D-lists same as before, but we build more than one A-list at each layer 1 node. At layer 1 node n_i we build an Al- and an Ar-list for each ancestor z of n_i by copying the top B y-points from the left and the right siblings that are descendants of z. We therefore build $O(\log(\frac{n}{B}))$ Al- and Ar-lists at n_i; each of these lists contains $O(B\log(\frac{n}{B}))$ points. The total number of disk blocks required to store all these Al- and Ar-lists at layer 1 is therefore $O(\frac{n}{B}/(\log^3(\frac{n}{B}))) \times O(\log^2(\frac{n}{B})) = O(\frac{n}{B}/(\log(\frac{n}{B})))$ which is the same as before. We continue in this manner, constructing multiple A-lists at each layer. A proof

similar to Lemma 2.1 shows that the space is still $O(\frac{n}{B})$ and the query time is $O(\log_B n + \log^*(\frac{n}{B}) + t/B)$ I/O's.

3.4 Improving the time required to answer queries: We can improve the time required to answer 2- and 3-sided queries by using the sampling idea once more. We build a cache that samples from its ancestors in the previous $O(\log^*(\frac{n}{B}))$ layers when we reach sets of size $O(B(\log^*(\frac{n}{B}))^3)$. Continuing further we again build these sampling structures when we reach sets of size $O(B(\log^* \log^*(\frac{n}{B}))^3)$. In this manner we can get a data structure that answers queries in $O(\log_B n + \mathcal{IL}^*(\frac{n}{B}) + t/B)$ I/O's. The details are omitted in this abstract. We therefore get the bounds of Theorem 3.1.

4 Data structures for answering general 2-dimensional queries

In this section we address the problem of answering general 2-dimensional queries. By using the data structure from Theorem 3.1 we show how to build a data structure with the following bounds.

THEOREM 4.1. *There exists a secondary memory data structure to answer 2-dimensional rectangular queries that requires $O(n \log(\frac{n}{B})/B)$ disk blocks of space. Queries can be answerd in $O(\log_B n + t/B + \mathcal{IL}^*(B))$ I/O's and updates can be made in $O(\log n(\log_B n + (\log_B n)^2/B))$ amortized disk I/O's per insertion/deletion.*

In Subsection 4.3 we show how to construct a data structure for answering 2-dimensional rectangular queries that requires only $O(n \log(\frac{n}{B})/B \log(\log_B(\frac{n}{B})))$ which is optimal.

P-range trees can also be used to answer range queries in main memory (we just set the value of B to be 2). In particular, if we are given a collection P of n points such that their x-values are in the range $[1..N]$ then we can build a data structure with the following bounds.

THEOREM 4.2. *There exists a main-memory static data structure for answering general 2-dimensional queries on P that uses $O(n \log n + N)$ space. The data structure can be used to answer any orthogonal 2-dimensional query q in $O(\mathcal{IL}^*(n) + t)$ time where t denotes the number of points of P that belong to q.*

The data structure of Theorem 4.2 can be used to build a multidimensional range-searching data structure for dimension three or more. By using a normalization technique due to Karlsson and Overmars [14,25] we can reduce the general d-dimensional range searching problem (with arbitrary coordinates) to one in which all the coordinates have value in the range $[1..n]$. This reduction only adds a factor of $d \log n$ to the query time. Therefore, using the p-range tree and the d-dimensional range searching structure of Willard and Leuker [11] we get the following bounds for answering general d-dimensional queries.

THEOREM 4.3. *Given a set of n points in d-dimensional space ($d \geq 3$), there exists a main memory data structure to store the points using $O(n \log^{d-1} n)$ storage such that orthogonal range queries can be performed in time $O(t + \mathcal{IL}^*(n) \log^{d-2} n)$.*

The data structure of Theorem 4.3 improves on the space-time product of previous data structures for d-dimensional range searching. The previous best data structures either used more space or required more time. For example the data structure due to Overmars [25] uses $O(n \log^{d-1} n)$ space (which is the same as ours) but requires $O(t + \log^{d-2} \log \log n)$ time to answer queries. The data structure by Chazelle [5] uses only $O(n \log^{d-2+\epsilon} n)$ space but requires $O(t + \log^{d-1} n)$ time. Here ϵ is an arbitrary constant. Thus our space-time product is better by a factor of $\log^{\epsilon} n/(\mathcal{IL}^*(n))$ than Chazelle's [5] structure and by a factor of $\log \log n/(\mathcal{IL}^*(n))$ than Overmars' [25] structure.

The rest of the section is organized as follows: In Subsection 4.1 we show how to use p-range trees to build a data structure for general 2-dimensional queries. In Subsection 4.2 we briefly describe how to adapt our data structures to main memory, and in Subsection 4.3 we show how to reduce the space requirements of our data structure.

4.1 A data structure for general 2-dimensional queries: We now show how to make use of the p-range tree to answer rectangular queries on a given point set P. Our data structure for answering rectangular queries will require $O(n \log(\frac{n}{B})/B)$ disk blocks of storage and will answer any query q in $O(\log_B n + \mathcal{IL}^*(B) + t/B)$ I/O's. In Subsection 4.3 we show how make our data structure space-optimal without increasing the query time.

To build our data structure we first build a search-tree R that divides the points in P in terms of their x-values. The leaves of R contain $\leq B$ points each. Each leaf α copies its B points on to all the nodes along the α-to-root path in R. Thus R is just a range-tree in two-dimension. Since each point is copied $O(\log(\frac{n}{B}))$ times there are $O(n \log(\frac{n}{B}))$ points in all. At each internal node z in R we build a p-range tree on the points copied at z. Since there are $O(n \log(\frac{n}{B}))$ copies, the total space required for all the p-range trees is $O(n \log(\frac{n}{B})/B)$.

Given a rectangular query $q = \{a \leq x \leq b, c \leq y \leq d\}$ we first find the top node z of the query q in R. This can be done by first finding the two corners $c1 = (a, c)$ and $c2 = (b, c)$ and locating the meeting-point of the

root-to-$c1$, and the root-to-$c2$ path. Let z_1 and z_2 be the children of z in R. Since R is a range tree it can be shown that in order to answer the query q we only need to answer two 3-sided queries on the points stored at z_1 and z_2. In particular, q is just the union of the 3-sided query $q_1 = \{a \leq x, c \leq y \leq d\}$ at z_1 and the 3-sided query $q_2 = \{x \leq b, c \leq y \leq d\}$ at z_2. We can use the p-range trees at z_1 and z_2 to answer these queries efficiently. By Theorem 3.1 we can find the points in q with $O(\log_B n + \mathcal{IL}^*(B) + t/B)$ I/O's. We therefore get the bounds of Theorem 4.1 for answering 2-dimensional rectangular queries.

4.2 Answering range queries in primary memory: The p-range tree can be implemented in main memory in a straightforward manner by setting the block size B to be 2. In order to get the bounds of Theorem 4.2 we augment the data structure with an array C of size N on the x-coordinates so that the corners of any given query can be found in $O(1)$ time by looking at the relevant entries of C. Using this and the data structure due to Harel and Tarjan [10] (see also [31]) for finding least common ancestors in $O(1)$ time we can get the bounds of Theorem 4.2 and Theorem 4.3.

4.3 An optimal-space data structure for answering 2-dimensional queries: Using techniques from [4], we can obtain an optimal space data structure that can answer any general 2-dimensional query in $O(\log_B n + t/B + \mathcal{IL}^*(B))$ disk I/O's. We do this by first creating a data structure for the point enclosure problem which is defined as follows: Given a set of input intervals on the x axis, report the set of intervals that contain a query point q. Chazelle [4] develops an optimal main memory algorithm for this problem using the idea of filtering search. We can adapt this data structure to secondary storage to get the following proposition.

PROPOSITION 4.4. There exists a data structure that uses $O(n/B)$ disk blocks to store an input set of n intervals such that any point enclosure query can be answered in $O(\log_B n + t/B)$ disk I/O's.

The general 2-dimensional range searching problem can now be tackled by considering it as a combination of at most two 3-sided queries and $\log_B n$ 2-sided queries. Using the p-range tree structure for the 3-sided queries and an adaptation of the data structure in Proposition 4.4 for the 2-sided queries, we get the following theorem. Details will be given in the full version.

THEOREM 4.5. *Given a set $P = \{p_1, p_2, \cdots, p_n\}$ of n points on the plane there exists a data structure for answering general 2-dimensional rectangular queries that requires $O((n/B)\log n/\log\log_B n)$ disk blocks of space. Using the data structure we can answer any rectangular query q in $O(\log_B n + t/B + \mathcal{IL}^*(B))$ I/O's where t denotes the number of points of P that belong to q.*

5 Lower bounds for general 2-dimensional range searching in secondary memory

In this section we prove a lower bound on the amount of disk-space needed to build an external-memory data structure for "efficiently" answering 2-dimensional orthogonal queries. Let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of points on the plane. We are interested in constructing an external-memory data structure that can answer any 2-dimensional rectangular query $q = \{a \leq x \leq b, c \leq y \leq d\}$ with $a(\log_B^c n + t/B)$ I/O's. Here, $a \geq 1$ and $c \geq 1$ are arbitrary constants and t is the number of points in P that belong to the query q. We show that in a fairly general purpose model of computing, any data structure that that answers rectangular queries with the worst-case efficiency of $O(\log_B^c n + t/B)$ I/O's, must use $\Omega(\frac{n}{B}\log(\frac{n}{B})/(\log\log_B n))$ disk blocks of storage.

Our model of computation called the *external memory pointer machine* is a natural generalization of a pointer machine [34], and is suitable for analyzing external memory algorithms. We model our data structure as a directed graph $G = (V, E)$ with a source node s. Each node of the graph corresponds to a disk block of memory and is allowed to have B data and pointer fields. In this way, the external memory can be modeled as a directed graph with outdegree at most B. The points of P are stored in the nodes of G. There may be multiple copies of a point stored in different nodes. The only restriction is that any node contain at most B points. For ease of exposition we will assume that many points in P can have the same x and y coordinates. The same lower bound can be proved without such a restriction.

Given a query q, the answering algorithm traverses G starting at the source node s, examining the points stored at the nodes that are visited. The searching algorithm must obey the following restrictions:

1. No node except the source s can be visited unless a node pointing to it has already been visited.
2. For every point p that belongs to the query q some node in G that contains p must be visited.

As the algorithm runs, it collects the nodes visited during the traversal in the set W. Initially W is set to be $\{s\}$. During the course of its run the algorithm is allowed to add and delete edges between nodes that are in W as long as the outdegree of each node does not exceed B. The query time is measured as $|W|$, the number of nodes visited.

Since we are interested in an algorithm that answers

queries with at most $a(\log_B^c n+t/B)$ I/O's, in our model $|W|$ must be less than or equal to $a(\log_B^c n + t/B)$. Consider a query q such that $t \geq B\log_B^c n$ (there are at least $B\log_B^c n$ points in P that belong to q). For such a query $|W| \leq 2at/B$. Suppose $t = B\log_B^c n$ then $|W| \leq 2a\log_B^c n$. Let us label a node v of W to be a *full node of* q if it contains at least $B/4a$ points that belong to q. Since each node contains at most B points there are at least $\log_B^c n/2$ full nodes in W.

To prove our lower bound we will generate a set S of hard queries. This set is similar to the one used by Chazelle [6] in proving a lower bound for this problem for main memory data structures. In particular we have the followin lemma.

LEMMA 5.1. *There is a set P of n points and a collection S of $\Omega(\frac{n}{B}\log(\frac{n}{B})/(\log_B^c n \log\log_B n))$ queries such that each query q in S has $B\log_B^c n$ points of P and no two queries $q_1, q_2 \in P$ share more than $B/8a$ points in common.*

Proof. The lemma can be proved by a modification of the method due to Chazelle [6]. Details can be found in [33].

Consider a set S of queries as described in Lemma 5.1 and let q_1 and q_2 be two queries in S. Let F_1 and F_2 be the set of full nodes of q_1 and q_2 respectively. Since q_1 and q_2 share no more than $B/8a$ points by Lemma 5.1, $F_1 \cap F_2$ is empty. Thus the full nodes of all the queries in S are distinct. Furthermore, by the discussion above each full node set contains at least $\log_B^c n/2$ full nodes. Therefore, the number of nodes in G is at least $|S|\log_B^c n/2$. Lemma 5.1 shows that the size of S is $\Omega(\frac{n}{B}\log(\frac{n}{B})/(\log_B^c(\frac{n}{B})\log\log_B n))$, we therefore get the following theorem.

THEOREM 5.1. *The data structure G must contain $\Omega(\frac{n}{B}\log(\frac{n}{B})/(\log\log_B n))$ nodes in order to answer queries in $O(\log_B^c n + t/B)$ I/O's in the worst-case.*

Theorem 5.1 shows that 2-dimensional range searching is harder to do in secondary memory is harder than it is in main memory. Since the optimum main memory data structure requires only $O(n\log n/\log\log n)$ space one would expect the optimum secondary memory data structure to require only $O(\frac{n}{B}\log n/\log\log n)$ disk blocks of storage. However, by Theorem 5.1 any mapping to secondary memory must incur an $\Omega(\log\log n/\log\log_B n)$ factor overhead in storage. Let us consider the common case when $B = \Omega(n^\epsilon)$ for some constant $\epsilon < 1$. In this scenario any secondary memory data structure must use $\Omega(\frac{n}{B}\log n/)$ disk blocks of storage while the main memory data structure requires only $O(n\log n/\log\log n)$ storage.

6 Conclusions and open problems

In this paper we have described a new data structure, the p-range tree, for performing 2-dimensional range searching in secondary memory. We have also shown how to use the p-range tree to get better data structures for higher dimensional range searching in main memory. Finally we have given some lower bounds that can be used show to that general 2-dimensional range searching is inherently harder to do in secondary memory than it is to do in primary memory.

Given the importance of multidimensional range searching to numerous database applications, we feel that understanding the implications of implementing range searching data structures in secondary memory is of both theoretical and practical importance. Our study of range searching data structures is far from complete. In particular, we would like to resolve the small gap of $\mathcal{IL}^*(B)$ between the lower and upper bounds for 2-dimensional range searching. Also, the complexity of range searching in secondary memory for dimensions higher than 2 remains open.

References

[1] R. Bayer and E. McCreight, "Organization of Large Ordered Indexes," *Acta Informatica* 1 (1972), 173–189.

[2] J. L. Bentley, "Algorithms for Klee's Rectangle Problems," Dept. of Computer Science, Carnegie Mellon Univ., unpublished notes, 1977.

[3] J. L. Bentley, "Multidimensional Divide and Conquer," *CACM* 23(6)(1980), 214–229.

[4] B. Chazelle, "Filtering Search: A New Approach to Query-Answering," *Siam J. of Computing* 15(3) (1986), 703–724.

[5] B. Chazelle, "A Functional Approach to Data Structures and its use in Multidimensional Searching," *SIAM J. of Computing* 17(3)(1988), 427–462.

[6] B. Chazelle, "Lower Bounds for Orthogonal Range Searching: I. The Reporting Case," *J. ACM* 37(2) (1990), 200–212.

[7] Y.-J. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," *Proceedings of IEEE, Special Issue on Computational Geometry* 80(9)(1992), 362–381.

[8] E. F. Codd, "A Relational Model for Large Shared Data Banks," *CACM* 13(6)(1970), 377–387.

[9] D. Comer, "The Ubiquitous B-tree," *Computing Surveys* 11(2)(1979), 121–137.

[10] D. Harel and R. E. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors," *SIAM J. of Computing* 13(1984), 338–355.

[11] D. E. Willard and G. S. Leuker, "Adding Range Restriction Capability to Dynamic Data Structures," *Journal of the ACM* 32(1985), 597–617.

[12] H. Edelsbrunner, "A New Approach to Rectangle Intersections, Part I," *Int. J. Computer Mathematics* 13(1983), 209–219.

[13] H. Edelsbrunner, "A New Approach to Rectangle Intersections, Part II," *Int. J. Computer Mathematics* 13(1983), 221–229.

[14] R. G. Karlsson and M. H. Overmars, "Normalized Divide and Conquer," Department of Computer Science, University of Utrecht, Technical Report RUU-CS-86-18, 1986.

[15] O. Günther, "The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases," *Proc. of the Fifth Int. Conf. on Data Engineering* (1989), 598–605.

[16] Antonin Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM-SIGMOD Conference on Management of Data* (1985), 47–57.

[17] C. Icking, R. Klein, and T. Ottmann, *Priority Search Trees in Secondary Memory (Extended Abstract)*, Lecture Notes In Computer Science #314, Springer-Verlag, 1988.

[18] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz, "Constraint Query Languages," *Proc. 9th ACM PODS* (1990), 299–313, invited to the special issue of *JCSS* on Principles of Database Systems (to appear). A complete version of the paper appears as Technical Report 90-31, Brown University.

[19] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter, "Indexing for Data Models with Constraints and Classes," *Proc. 12th ACM PODS* (1993), 233–243, invited to the special issue of *JCSS* on Principles of Database Systems (to appear). A complete version of the paper appears as Technical Report 93-21, Brown University.

[20] W. Kim and F. H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, 1989.

[21] D. B. Lomet and B. Salzberg, "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance," *ACM Transactions on Database Systems* 15(4)(1990), 625–658.

[22] E. M. McCreight, "Priority Search Trees," *SIAM Journal of Computing* 14(2)(1985), 257–276.

[23] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems* 9(1)(1984), 38–71.

[24] J. A. Orenstein, "Spatial Query Processing in an Object-Oriented Database System," *Proc. ACM SIGMOD* (1986), 326–336.

[25] M. H. Overmars, "Efficient Data Structures for Range Searching on a Grid.," Department of Computer Science, University of Utrecht, Technical Report RUU-CS-87-2, 1987.

[26] J. A. La Poutré, "Dynamic Graph Algorithms and Data Structures," Department of Computer Science, University of Utrecht, Ph. D. Thesis.

[27] S. Ramaswamy and S. Subramanian, "Path Caching: A Technique for Optimal External Searching," *Proc. 13th ACM PODS* (1994), 25–35.

[28] J. T. Robinson, "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. ACM SIGMOD* (1984).

[29] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1989.

[30] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, 1989.

[31] B. Schieber and U. Vishkin, "On Finding Lowest Common Ancestors: Simplification and Parallelization," *Proc. 3rd Aegean Workshop on VLSI Algorithms and Architecture, Lecture Notes in Computer Science* 319(1988), 111–123.

[32] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. 1987 VLDB Conference, Brighton, England* (1987).

[33] S. Subramanian and S. Ramaswamy, "The P-range tree: A new data structure for range searching in secondary memory," *Technical Report Brown University* (1994).

[34] R. E. Tarjan, "A Class of Algorithms that Require Nonlinear Time to Maintain Disjoint Sets," *JCSS* 18(1979), 110–127.

[35] S. Zdonik and D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1990.