Journal of Universal Computer Science, vol. 10, no. 9 (2004), 1325-1353
submitted: 21/1/01, accepted: 21/2/04, appeared: 28/9/04 © J.UCS

Geometric Retrieval for Grid Points in the RAM Model

Spyros Sioutas, Christos Makris, Nektarios Kitsios, George Lagogiannis, John Tsaknakis, Kostas Tsichlas, Bill Vassiliadis
Department of Computer Engineering and Informatics, University of Patras,Greece
{sioutas,makri,kitsios,lagogian,tsaknaki,tsihlas,bb}@ceid.upatras.gr

Abstract: We consider the problem of d-dimensional searching ($d \geq 3$) for four query types: *range*, *partial range*, *exact match* and *partial match searching*. Let N be the number of points, s be the number of keys specified in a partial match and partial range query and t be the number of points retrieved. We present a data structure with worst case time complexities $O(t + \log^{d-2} N)$, $O(t + (d-s) + \log^{s} N)$, $O(d + \sqrt{\log N})$ and $O(t + (d-s) + s\sqrt{\log N})$ for each of the aforementioned query types respectively.

We also present a second, more concrete solution for exact and partial match queries, which achieves the same query time but has different space requirements. The proposed data structures are considered in the RAM model of computation.

Key Words: Multi-Dimensional Data Structures, Indexing, Spatial Data, Grid Model, File Systems
Category: E.1, E.5

1 Introduction

A geometric retrieval system must respond to a query by initiating a search operation, followed by the retrieval of all objects requested by the query. The objective of this paper is to propose efficient data structures, in the RAM model of computation, that facilitate the operations of geometric retrieval. In particular, we consider the operations of *range searching*, *partial range searching*, *exact and partial match*.

Let S be a collection of N points, each of which is an ordered d-tuple $(r_1, \ldots, r_d)$ of values r_i, $1 \leq i \leq d$. Each component of the d-tuple is called an *attribute* or a *key*. A retrieval request is the specification of certain conditions that must be satisfied by the keys of the retrieved records. The queries considered, are categorized as follows:

- *Range query* specifies d ranges, one for each key.
- *Partial range query* specifies $s < d$ key ranges, with the remaining $d - s$ unspecified.
- *Exact match query* specifies an exact value for each key.
- *Partial match query* specifies $s < d$ key values, with the remaining $d - s$ ones unspecified.

The last two types of queries can are special cases of the general range query, if we allow the exact range $[x, x]$ and the infinite range $(-\infty, \infty)$, as a possible query range for each key. As a result, an exact match query can be represented as a general range query, in which all d ranges are exact. In the same manner, a partial match query has only s exact ranges.

In this paper we consider the unit-cost RAM with a word length of w bits, which models what we program in imperative programming languages such as C. The words of the RAM are addressable and these addresses are stored in memory words, imposing that $w \geq \log N$. As a result, the universe U consists of integers in the range $0 \ldots 2^w - 1$. It is also assumed that the RAM can perform the standard AC^0 operations of addition, subtraction, comparison, bitwise Boolean operations and shifts, as well as multiplications in constant worst-case time on $O(w)$-bit operands. One of the basic features of the RAM is that the content of the elements are used for addressing, which is one of the basic differences with other comparison based models (such as PM).

The problem of range searching is one of the most extensively studied problems in computational geometry. In the static case, a set of N points in the d-dimensional space is given and the goal is to preprocess them in a data structure, so that for any query range, the points inside this range can be found efficiently. Numerous solutions for this problem have been suggested (see the survey paper of Agarwal [4]). In the RAM model of computation, the best solution [12] needs $O(\log^{d-2} N + t)$ time to retrieve the output and uses $O(N \log^{d-1} N)$ space. In the Pointer Machine model of computation, the best general solution is based on the layered range tree [19]. This is a modified version of the range tree, which exploits the benefits of the fractional cascading technique, and requires $O(n \log^{d-1} N)$ storage and $O(\log^{d-1} N)$ query time.

In the 2-dimensional case, assuming that the query ranges are of the form $[a, b] \times (-\infty, c]$, the problem can be solved optimally in $O(N)$ space and $O(\log N + t)$ time, by using the priority search tree of McCreight [18]. If the stored points have integer coordinates in the range $[1, M] \times R$ (the *grid assumption*), then more efficient solutions can be obtained. Overmars [20], has presented a structure with $O(\log \log M + t)$ time and $O(N)$ space, but with a very expensive preprocessing phase. In the same paper, he has presented an alternative solution with $O(\sqrt{\log M} + t)$ query time, $O(M)$ space and $O(N \log N)$ preprocessing time. A different solution with $O(\log \log M + t)$ query time and $O(M + N)$ space and preprocessing time, has been presented by Fries et al. [10]. All these solutions rely on modifications and extensions of the priority search tree structure. Note that in the special case in which the query ranges are of the form $(-\infty, a] \times (-\infty, b]$, a more efficient solution can be obtained by using the window list structure of Chazelle [6]. This solution has $O(N + M)$ space and $O(t)$ query time.

However, largely unnoticed has remained the solution of Gabow, Bentley and

Tarjan [12], which can be used to solve the problem in $O(N + M)$ space and $O(t)$ query time. Their structure relies on the Cartesian Tree of Vullemin [23], which is in fact the predecessor of the priority search tree. They make use of the machinery developed in [15] (see also [21]) for answering *lowest common ancestor* (lca) queries for two nodes of an arbitrary tree in $O(1)$ time. Their algorithm retrieves each point in the query range, by executing an lca query. Although the time complexity of the lca queries is $O(1)$, their implementation is quite complicated. As a result, there is a considerable time overhead for each retrieved point.

In this paper, we present a modified priority search tree, which matches the performance of the structure of Gabow et al. [12]. The overall solution requires at most *two* lca queries for each range query and not for every point of the answer (as the solution in [12]). Consequently, although our new structure uses other complicated techniques (persistent lists and microset table lookup), it is faster that the solution of [12]. This claim is validated both theoretically and experimentally.

We also present a d-dimensional data structure, which uses as its skeleton a range tree ([19]), with its last level being replaced by our modified priority search tree. As a result, we achieve in a more simple and concrete way, $O(t+\log^{d-2} N)$ response time for d-dimensional range queries. Consequently, we perform partial range queries in $O(t+(d-s)+\log^{s} N)$ time. Furthermore, an exponential search tree [2] is used as an additional index, for each of the d dimensions of the problem. In this way, exact match and partial match queries are improved, achieving $O(t + \sqrt{\log N})$ and $O(t + (d - s) + s\sqrt{\log N})$ query times respectively. These trees are used to speed up both exact match and partial match queries, using a set of pointers to the skeleton structure. Exponential search trees are multiway trees that answer efficiently queries in one-dimensional space. Their combination with a number of other techniques such as the Fusion Tree technique [11] and the van Emde Boas trees [24], result in very fast search structures. Furthermore, the tuples stored in these additional trees are also stored in the subtrees of the skeleton tree itself, thus duplicating the data stored. Nevertheless, this technique helps when partial queries are performed. In order to achieve a more concrete structure, we give a second solution for the same problem by incorporating exponential search trees in the previous structure. In this way, space requirements are reduced in the average case (for example in typical database files), without affecting the retrieval times.

The remainder of the paper is organized as follows. In Section 2 the priority search tree is described. In Section 3 we present our modified priority search tree. In Section 4 we show how we match the $O(t+\log^{d-2} N)$ time, in a simpler and more concrete manner. In Section 5 we present the first solution of our fast exact and partial match search algorithm while in Section 6 we give a second

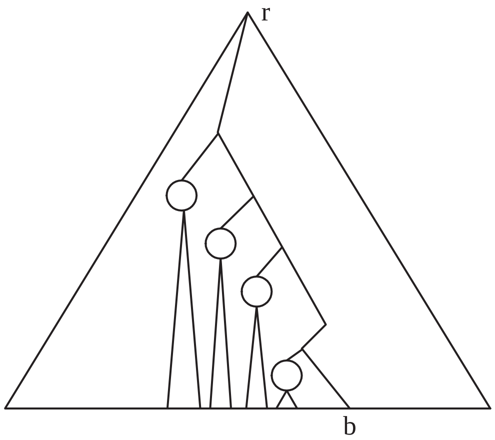

Figure 1: P_b: the search path, L_b: the nodes that are left sons of nodes on P_b and do not belong to the path.

solution for the same problem. We conclude in Section 7.

2 Preliminaries

In this section we briefly review the priority search tree of McCreight [18]. Let S be a set of N points on the plane. We want to store them in a data structure, so that the points that lie inside a semi-infinite strip of the form $[a, b] \times (-\infty, c]$, can be found efficiently.

The priority search tree is a binary search tree over the x-coordinates of the points with logarithmic depth. The root of the tree contains the point p with the minimum y-coordinate. The left (resp. right) subtree is recursively defined for the set of points in $S - \{p\}$. The set $S - \{p\}$ is partitioned equally into the two subtrees of the root. As a result, it is easy to see, that a point is stored in a node on the search path from the root to the leaf containing its x-coordinate.

Initially, queries with ranges that are half-infinite in both x and y directions are considered (i.e. they are of the form $(-\infty, b] \times (-\infty, c]$). This special case is also known as *quadrant range search*. To answer a quadrant range query, we find the $O(\log N)$ nodes in the search path P_b for point b. Let L_b be the left children of these nodes that do not lie on the path (see Figure 1). In $O(\log N)$ time, the points of the nodes of $P_b \bigcup L_b$ that lie in the query-range can be determined. Then, for each node of L_b storing a point inside the range query, its two children are visited and checked whether their points lie in the range. This procedure continues recursively, as long as points in the query-range are found.

The correctness of the query algorithm is proved as follows. First, observe that nodes to the right of the search path, have points with x-coordinate larger than b and therefore lie outside the query-range. The points of P_b may have x-coordinate larger than b or they may have y-coordinate larger than c. In any case, they are not reported. The nodes of L_b and their descendants have points with x-coordinate smaller than b, so that only their y-coordinates need to be tested. The children of nodes of L_b with y-coordinate less than c must be considered. In particular, the reporting procedure proceeds recursively, as long as points inside the query range are found. If a point of a node u does not lie inside the query-range, then this point has y-coordinate larger than c. Therefore, all points in the subtree rooted at u lie outside the query-range and they are not reported. We can easily bound the query time by $O(\log N + t)$, since $O(\log N)$ time is needed to visit the nodes in $P_b \bigcup L_b$ and $O(t)$ time is necessary for the reporting procedure in their subtrees.

Finally, consider the general case where the query ranges are of the form $[a, b] \times (-\infty, c]$. The query algorithm finds the nodes in the two search paths P_a and P_b for a and b respectively. Let C be the set of nodes, consisting of all left children of the nodes in P_a and all right children of the nodes in P_b. The nodes of $P_a \bigcup P_b \bigcup C$ that have points inside the query range can be determined in $O(\log N)$ time. Then, the descendants of nodes of C are traversed recursively as long as their points lie in the query range. Note that the nodes of C and their descendants have x-coordinates inside the query-range and as a result only their y-coordinates need to be considered. The correctness of the algorithm follows by similar arguments as in the case of quadrant range queries, while the query time is $O(\log N + t)$.

3 The Modified Priority Search Tree

3.1 Quadrant Range Search on a Grid

Let S be a set of N points on the plane with coordinates in the range $[1, M] \times R$. Without loss of generality we assume that all points are distinct. We will show how to store the points in a data structure, so that the t points in a query range of the form $(-\infty, b] \times (-\infty, c]$, can be found in $O(t)$ time. Our structure relies on the priority search tree, which we augment with list-structures similar to those in [20].

We store the points in a priority search tree T, of size $O(N)$ as described in the previous section. For convenience we will assume that the tree T is a complete binary tree (i.e. all its leaves have depth $\log N$). Note that if N is not a power of 2, then we may add some dummy leaves so that T becomes complete. We also use an array A of size M, which stores pointers to the leaves of T. Specifically, $A[i]$ contains a pointer to the leaf of T with maximum x-coordinate

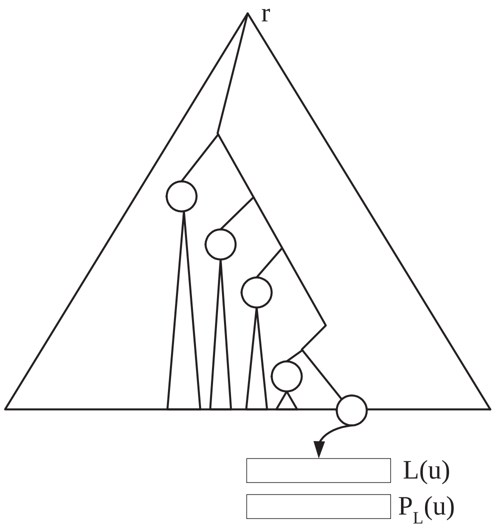

Figure 2: $L(u)$: it stores the points of the nodes of L_i. $P_L(u)$: it stores the points of the nodes of P_i which have x-coordinate smaller or equal to i.

smaller or equal to i. This array is used to determine in $O(1)$ time the leaf of the search path P_b for b. In each leaf u of the tree with x-coordinate i we store the lists $L(u)$, $P_L(u)$. The list $L(u)$ stores the points of the nodes of L_i. The list $P_L(u)$ stores the points of the nodes of P_i which have x-coordinate smaller or equal to i. Both lists also contain pointers to the nodes of T that contain these points. Each list $L(u)$, $P_L(u)$, stores its nodes in increasing y-coordinate of their points (see Figure 2).

To answer a query with a range $(-\infty, b] \times (-\infty, c]$ we find in $O(1)$ time the leaf u of the search path P_b for b. Then we traverse the list $P_L(u)$ and report its points until we find a point with y-coordinate greater than c. We traverse the list $L(u)$ in the same manner and find the nodes of L_b whose points have y-coordinate less than or equal to c. For each such node we report its point and then we continue the reporting procedure further in its subtree, as long as we find points inside the range.

The following theorem bounds the size and query time of our structure.

Theorem 1. *Given a set of N points on the plane with coordinates in the range $[1, M] \times R$ we can store them in a data structure with $O(M + N \log N)$ space that allows quadrant range queries to be answered in $O(t)$ time, where t is the number of reported points.*

Proof. The query algorithm finds the t' points of nodes of $P_b \bigcup L_b$ that lie inside the query-range in $O(t')$ time by simple traversals of the lists $P_L(u)$, $L(u)$. The search in subtrees takes $O(t)$ additional time for reporting t points in total. Therefore, the query algorithm needs $O(t)$ time. Each list $P_L(u)$, $L(u)$ stores respectively points in the nodes of a path, and points in the left children of nodes of a path. So the size of each list is $O(\log N)$ and the space of T is $O(N \log N)$. The space of the whole structure is $O(M + N \log N)$ because of the size of the array A.

The $O(N \log N)$ term in the space bound is due to the size of the lists $P_L(u)$ and $L(u)$. We can reduce the total space of these lists to $O(N)$ by making them persistent. Ordinary structures are ephemeral in the sense that update operations make permanent changes to the structures. Therefore in ordinary structures it is impossible to access their old versions (before the updates). According to the terminology of Driscoll et al. [8] a structure is persistent, if it allows access to older versions of the structure. There are two types of persistent data structures: partially and fully persistent. A partially persistent data structure allows updates of its latest version only, while a fully persistent one allows updates of any of its versions. In [8], a general methodology is proposed for making data structures of bounded in-degree persistent. With their method such a structure can be made partially persistent with $O(1)$ amortized space cost per change in the structure. In our case a list can be made partially persistent with a $O(1)$ amortized increase in space per insertion/deletion.

We show how to implement the lists $P_L(u)$ using a partially persistent list. Let u be a leaf in T and let w be its predecessor (the leaf on the left of u). We denote by x_u the x-coordinate of u and by x_w, the x-coordinate of w. The two root-to-leaf paths P_{x_u}, P_{x_w}, share the nodes from the root of T to the nearest common ancestor of u, w. As a result, we can create $P_L(u)$ by updating $P_L(w)$ in the following way. First we delete from list $P_L(w)$ the points that don't lie on P_{x_u}. Then we insert the points of P_{x_u} which have x-coordinate smaller or equal to x_u. In this way we can construct all lists as versions of a persistent list: we begin from the leftmost leaf and construct the list $P_L(u)$ of each leaf u by updating the one of its predecessor (see Figure 3).

The total number of insertions and deletions is $O(N)$ because each point is inserted and deleted only once. Therefore the space of all the lists is $O(N)$. In the same way, lists $L(u)$ are constructed for all leaves in $O(N)$ space. Therefore:

Theorem 2. *Given a set of N points on the plane with coordinates in the range $[1, M] \times R$ we can store them in a data structure with $O(N+M)$ space that allows quadrant range queries to be answered in $O(t)$ time, where t is the number of answers.*

The preprocessing time is $O(M + N \log N)$ but with a more careful implementation we can reduce this complexity to $O(M + N)$, by using the pruning

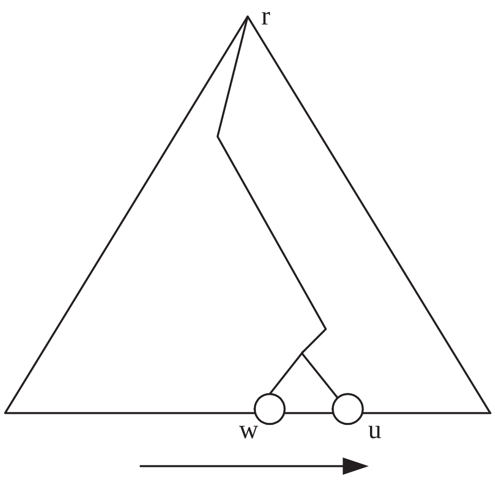

Figure 3: Lists $P_L(u)$ and $L(u)$ are implemented as partially persistent lists, by performing a sweep from left to right.

technique as in [10]. We will discuss this solution in the next subsection, in which we show how to handle query ranges of the form $[a, b] \times (-\infty, c]$.

3.2 Three-Sided Range Queries

We will show how to solve the problem, when the queries are of the form $[a, b] \times (-\infty, c]$. That is, we want to find the points of S that lie inside a query rectangle, which extends infinitely in the negative y-direction. First we present a simple solution with $O(t)$ time and $O(M + N \log N)$ space and then we show how to reduce the space to $O(N + M)$.

3.2.1 A Structure With $O(t)$ Time

We will use a modified priority search tree T of size $O(N)$ which stores the N points of S. We denote by T_v the subtree of T with root v.

The tree structure T has the following properties:

- Each point of S is stored in a leaf of T and the points are in sorted x-order from left to right.
- Each internal node v of T stores a point $p(v)$ of S. The point $p(v)$ is the point with the minimum y-coordinate amongst the points stored in the leaves of T_v.

– Each node v is equipped with a secondary list $S(v)$. $S(v)$ contains the points stored in the leaves of T_v in increasing y-coordinate.

We also use an array A of size M which stores pointers to the leaves of T. Specifically $A[i]$ contains a pointer to the leaf of T with maximum x-coordinate smaller or equal to i. With this array we can determine in $O(1)$ time the leaves of the search paths P_a, P_b for a and b respectively. Let u be a leaf of the tree and let i be the x-coordinate stored in u. P_i denotes the search path for i, i.e. it is the path from the root to u. We denote by P_i^j the subpath of P_i with nodes of depth larger than j. Similarly L_i^j (respectively R_i^j) denotes the set of nodes that are left (resp. right) children of nodes of P_i^j and do not belong to P_i^j. Note that $0 \leq j \leq \log N$ so there are $\log N$ such sets P_i^j, L_i^j, R_i^j for each leaf u.

In leaf u we store the following lists L^j, R^j: The list $L^j(u)$ (resp. $R^j(u)$) stores the points $p(v)$ of the nodes v of L_i^j (resp. R_i^j) in increasing y-coordinate and pointers to the respective nodes. In each leaf we also store arrays of size $\log N$, so that for a given j any list $L^j(u)$, $R^j(u)$ can be accessed in constant time.

To answer a query with the range $[a,b] \times (-\infty, c]$ we use the array A to find the two leaves u, w of T in the search paths P_a, P_b respectively. A simple algorithm of Harel and Tarjan [15] is used to compute the depth j of the lowest common ancestor of u, w in $O(1)$ time. Then, we traverse $R^{j-1}(u)$, $L^{j-1}(w)$ and we find the nodes of R_a^{j-1}, L_b^{j-1} whose stored points have y-coordinate less than c. For each such node v we traverse the list $S(v)$ in order to report the points stored in the leaves of T_v that satisfy the query. The following lemma ensures the correctness of the procedure.

Lemma 3. *The query algorithm correctly computes the points of S inside the range $[a,b] \times (-\infty, c]$.*

Proof. Since all the points of S with x-coordinates between a,b are stored in the leaves of T between u,w it follows (by the construction of the modified priority search tree and the used query algorithm) that the algorithm reports all the points that lie in the query. Because only points that lie in the query range are reported, the algorithm is correct.

Theorem 4. *Given a set S of N points on the plane with coordinates in the range $[1,M] \times R$ we can store them in a data structure with $O(M + N \log N)$ space that allows three-sided range queries to be answered in $O(t)$ time, where t is the number of reported points.*

Proof. Let $t' \leq t$ be the number of points in the nodes of R_a, L_b that lie inside the query-range. The algorithm takes $O(t')$ time to find these points, by traversing each corresponding list until a point outside the query range is found. Then in

$O(t)$ time the algorithm finds the rest of the points by searching further in the secondary lists of those nodes of R_a, L_b which have points inside the query range.

Each leaf stores $\log N$ lists L^j, R^j. By implementing these list as partially persistent sorted lists (as in the proof of Theorem 2), the total space of these lists is $O(N \log N)$. Also each point of S is stored in $O(\log N)$ secondary lists. Consequently, the space of the tree structure is $O(N \log N)$. We also need $O(M)$ space for the array A.

3.2.2 Reducing the Space of the Structure

To reduce the space of the structure the technique of *pruning* is used as in [10, 20]. However, pruning alone does not reduce the space to linear. We can get better but not optimal results by applying pruning recursively. To get an optimal space bound we will use a combination of pruning and table lookup.

The pruning method is as follows: Consider the nodes of T, which have height $\log \log N$. These nodes are roots of subtrees of T of size $O(\log N)$. There are $O(N/\log N)$ such nodes. Let T_1 be the tree whose leaves are these nodes and let T_2^i be the subtrees of these nodes for $1 \leq i \leq O(N/\log N)$. Note that T_1 has size $O(N/\log N)$. We call T_1 the first layer of the structure and the subtrees T_2^i the second layer. T_1 and each subtree T_2^i are structured as modified priority search trees.

We implement the tree T_1 and the subtrees T_2^i using the solution of the previous section. Tree T_1 needs space $O(N)$ according to Theorem 4 since its size is $O(N/\log N)$. Each subtree T_2^i has $O(\log N)$ leaves and depth $O(\log \log N)$. Therefore, it needs $O(\log N \log \log N)$ space. The total size of subtrees in the second layer is $O(N \log \log N)$. As a result, the space used is $O(N \log \log N)$ for the whole structure and $O(M)$ for the array.

To answer a query $[a, b] \times (-\infty, c]$, array A is used to find the leaves of the subtrees T_2^i , T_2^j on the search paths for a, b (note that both leaves may belong to the same subtree, i.e. i may be equal to j). We query both structures as in the previous section in $O(t_2)$ time, where t_2 is the number of the reported points. The roots of these subtrees are leaves of T_1. Consequently, we can query T_1 in the same manner and find, in $O(t_1)$ time, the t_1 points that lie inside the query range.

We may apply the same pruning method to the subtrees of the second layer and continue recursively. If we apply the method $k > 1$ times then the space of the structure will be $O(M + N \log^{(k)} N)$ and the query time is $O(k + t)$, where $\log^{(k)}$ is the composition of the log function k times. If we choose k to be constant then we can reduce the space, without affecting asymptotically the query time. The best space bound we can get is $O(M + N \log^* N)$ but then the query time will be $O(log^* N + k)$. We summarize in the following:

Theorem 5. *Given a set of N points on the plane with integer coordinates in the range $[1, M] \times R$, we can store them in a data structure with $O(M + kN + N \log^{(k)} N)$ space that allows three-sided range queries to be answered in $O(t+k)$ time, where t is the number of answers and k is an integer parameter $2 \leq k \leq \log^* N$.*

3.2.3 Achieving Optimal Time and Space Performance

In the previous subsection we showed how to reduce the space of our structure by recursive pruning. This process partitioned the modified priority search tree T in a number of layers. Each layer consists of a number of disjoint subtrees of T. The trees of the i-th layer ($i > 2$) have size $O(\log^{(i-1)} N)$ each. Note that if the structure has k layers, then each of the $k-1$ higher layers has total size $O(N)$, while the last layer has total size $O(N \log^{(k)} N)$. We need a more space-efficient way to implement the last layer.

We will use a three-layered structure. We implement the first two layers as in the previous section. The third layer consists of $O(N/\log\log N)$ modified priority search trees, each of size less than $\log\log N$. The technique of table lookup will be used to implement the modified priority search trees of this layer. This is reminiscent of the way Gabow and Tarjan implement the so-called microsets for the disjoint set union problem [13], so we accordingly will call the priority search trees of the last layer microtrees. Like the solution in [13], that encodes amounts of information in words of memory, we will assume that each memory cell can hold only integers in the range $[0, N]$ (i.e. each memory cell has $\log N$ bits).

The microtrees are complete binary trees with m leaves, where $m \leq \log\log N$ is a power of 2. Each microtree is a modified priority search tree, which stores at most m points. We will assume that each microtree stores exactly m points. If this is not the case we can add some dummy points with y-coordinate equal to $+\infty$. Note that the query algorithm will not report these dummy points. Let $p_1, p_2, \ldots, p_m$, be the points stored in the leaves of the microtree in increasing x-order. Let also $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$ be a permutation of $1, 2, \ldots, m$, so that $p_{\sigma(1)}, p_{\sigma(2)}, \ldots, p_{\sigma(m)}$ is the sequence of the points stored in the leaves of the microtree in increasing y-order. This permutation can be encoded using $m \log m \leq \log\log N \log\log\log N < \log N$ bits and therefore it can be stored right justified in one memory cell. Let k,l be two integers with $1 \leq k \leq l \leq m$ and $output(k, l) = p'_1, \ldots, p'_{l-k+1}$, be the points $p_k, \ldots, p_l$ in increasing y-coordinate. The list $output(k, l)$ can be uniquely identified with a $(l-k+1)$-tuple $\sigma'(k, l) = \sigma'(1), \ldots, \sigma'(l-k+1)$ such that $\sigma'(i) = j$ iff $p'_i = p_j$. This tuple can be encoded using at most $m \log m \leq \log\log N \log\log\log N < \log N$ bits and therefore it can be stored right justified in one memory cell. The crucial observation is that given σ,κ,l then the tuple $\sigma'(\kappa, l)$ is uniquely identified. Based on this observation we build a table $Answer\,[i, j, perm]$ where i,j range over all possible values between

1 and m, and perm ranges over all possible values between 1 and $2^{m \log m}$. If *perm* is a valid encoding of a permutation σ of a microtree and $i \leq j$ then the entry $Answer\,[i, j, perm]$ stores the value $\sigma'(i, j)$ otherwise it stores zero.

We also need another table W with $O(N/\log\log N)$ entries such that $W\,[i]$ stores a value corresponding to the encoding of the permutation of the i-th microtree (the microtrees of T are numbered from left to right with the values $1, \ldots, B$ with $B = O(N/\log\log N)$).

Given the above tables it is easy to answer a query $[a, b] \times (-\infty, c]$ for the i-th microtree. Initially, by using array A we locate the two leaves u and w of the microtree as in Section 2. Assume that these are the k-th and the l-th leaf of the microtree. Then in $O(1)$ time we access $s = Answer\,[k, l, W\,[i]]$ and we traverse the set of nodes represented by the word s until a point with greater y-coordinate than c is found. It is obvious that the search time is $O(t)$, where t the number of reported points. As a result, the time bound of the query algorithm of the previous section is not affected.

It remains to bind the size of the tables and the time needed for their construction.

Lemma 6. *All tables need $O(N)$ space and can be constructed in $O(N)$ time.*

Proof. Table W has size $B = O(N/\log\log N)$. The table $Answer$ has size $O(m^2 2^{mlogm})$ and each entry of the table needs $O(m)$ time to be computed. Since $m = O(\log\log N)$ the table $Answer$ needs linear space and can be constructed in linear time. We also need to compute the permutation σ for every one of the $B = O(N/\log\log N)$ microtrees and store the number that encodes it, in the table W. This can be done in $O(m) = O(\log\log N)$ time, for each microtree, if the y-order of its points is known. All these y-orders of the points of microtrees can be derived in $O(N)$ time, from the y-order of all the points of S (which is known since the points lie on a grid). Therefore the total time to compute the table Q is $O(N)$.

We summarize in the following:

Theorem 7. *Given a set of N points on the plane with integer coordinates in the range $[1, M] \times R$ we can store them in a data structure with $O(N + M)$ space that allows three-sided range queries to be answered in $O(t)$ time, where t is the number of reported points. The structure can be built in $O(N + M)$ time.*

3.2.4 Theoretical Comparison

In order to compare the solution based on the use of the *Cartesian Tree* with our approach, we will assume a standard RAM model of computation. The RAM can only read, write, address, add, subtract, multiply, and divide in constant time and only with numbers of size $O(\log M)$ bits. In this model of computation, bit-level operations dealing with $O(\log M)$ sized words can be simulated by table lookup (the construction of these tables can be performed during an initial preprocessing phase). We assume, in the sequel, that both algorithms will be compared by using the number of table lookups as a measure of comparison.

1st Solution (*Cartesian Tree*)

Let $S = \{(x_1, y_1), (x_2, y_2), , (x_N, y_N)\}$ be the set of stored points , $x_1 \leq x_2 \leq \ldots \leq x_N$. A Cartesian tree T for S, is a binary tree of N nodes with each node being labeled by a point in S. The tree T is defined recursively as follows: the root is labeled with (x_m, y_m), where $y_m = \min\{y_i | i = 1, \ldots, N\}$. Its left subtree is a Cartesian Tree for $\{(x_i, y_i) | i = 1, \ldots, m-1\}$ and its right subtree is a Cartesian Tree for $\{(x_i, y_i) | i = m+1, \ldots, N\}$. Let $p(v)$ be the point labeling the node v of T, $p_x(v)$ be its x-coordinate and $p_y(v)$ be its y-coordinate. In addition, the structure uses an array A of size M (M denotes the universe size), which stores pointers to the nodes of T. Specifically $A[i]$ contains a pointer to the node of T which is labeled with the point in S having maximum x-coordinate smaller or equal to i. The crucial property of the above construction is that given two nodes u and w of T with $p_x(u) = x_i$, $p_x(w) = x_j$, $i < j$, then $p_y(lca(u, w)) = \min\{y_t | i \leq t \leq j\}$, where the symbol $lca(u, w)$ defines the lowest common ancestor of nodes u,w on Cartesian Tree.

As a consequence, given a query range of the form $[a, b] \times (-\infty, c]$, we firstly use the array A to locate the two nodes u,w of T with $A[a] = u$, $A[b] = w$. Let $z = lca(u, w)$ with $p(z) = (x_i, y_i)$. If $y_i > c$ then the query algorithm halts, otherwise we report the point (x_i, y_i) and we continue recursively the same way, by probing $lca(u, u_1)$ and $lca(u_2, w)$, where u_1 is the predecessor (in symmetric order) of z in T (that is the point that is stored in u_1 is $p(u_1) = (x_i - 1, y_i - 1)$) and u_2 is the successor (in symmetric order) of z in T (that is the point that is stored in u_2 is $p(u_2) = (x_i + 1, y_i + 1)$). The time complexity of the above procedure (in terms of table lookups) is $2 + 2kt_{lca}$, where t_{lca} is the number of table lookups in order to perform a lowest common ancestor computation. In order to compute the lowest common ancestor of two nodes in T, we will use the algorithm described by Schieber and Vishkin [21] (this algorithm simplifies the approach described in Harel and Tarjan [15]). We begin by giving an outline of the solution described in [21] (our description is based on the book of Gusfield [14]).

After an initial (linear time) preprocessing phase the following numbers are computed for each node v of T:

- A number $d(v)$ denoting the number given to v, according to a depth first traversal of T.
- A number $h(v)$ denoting the position (counting from the right) of the least

significant 1-bit in the binary representation of $d(v)$.

- A number $I(v) = d(w)$ where w is the node in T, such that $h(w)$ is maximum over all nodes in the subtree of v (including v). This number encodes a mapping from nodes in T to a complete binary tree C of depth $\log N$. More specifically, the node v of T is mapped to the node of C, whose symmetric order number equals $I(v)$ (the intuition behind this mapping is explained in [15] and [14]).
- A binary number A_v. This number has size $O(\log N)$ bits. The bit $A_v(i)$ is set to 1, if and only if there is an ancestor of node v that maps to height i of C, (that is v has an ancestor u so that $h(I(u)) = i$).
- Moreover, a linear sized table $L()$ is computed. Each entry of the table $L()$ points to a node of T.

In order to compute the lowest common ancestor of two nodes v,w the algorithm follows the following steps:

1. It computes the lowest common ancestor b in C of nodes $I(v)$ and $I(w)$, (3 bit level operations).
2. It computes the smallest bit position greater than equal to $h(b)$ such that both A_v, A_w have 1-bits in this position (2 bit level operations). Let this position be k.
3. It computes nodes $v', w' \in T$, as follows (we present only the computation for locating node v', since the computation for node w' is analogous):
4. Locate the position l of the rightmost 1-bit in A_v (this information could be stored in v, computed during the initial preprocessing phase).
5. If $l = k$ then $v' = v$.
6. Otherwise ($l < k$) locate the position t of the left-most 1-bit in A_v that is to the right of position k (1 bit level operation). Form the number consisting of the bits of $I(v)$ to the left of the position t, followed by a 1-bit in position t, followed by all zeros (1 bit-level operation). Let i' be that number. Set node v' to be the parent of node $L(i')$ (1 table lookup).
7. If $v' < w'$ then v' is the lowest common ancestor of v, w else the nearest common ancestor of v, w is w'.

It is possible to replace the computation performed in steps 1,2 by two bit-level operations (see [14]). As a result, the lowest common ancestor computation needs 6 bit level operations and 2 table lookups. Consequently, the worst case number of table lookups is 8, and the complexity of the algorithm $2 + 16k$ table lookups.

2nd Solution (*Our Approach*)

Our three layered data structure used table precomputation and table lookups only for the third layer. Consider queries of the form $[a, b] \times (-\infty, c]$. Let k be the size of the output.

Initially, the query algorithm locates the two leaves in the structure that correspond to the two leaves a and b (2 table lookups). Then, the following two cases are considered:

1. The two leaves belong to the same microtree (let it be the i-th microtree) and let these be the k-th and the l-th leaves. Then with another two table lookups we get the word $s = Answer\,[k, l, W\,[i]]$). Then, we have to perform for every point that belongs to the answer (plus an extra point that does not belong to the answer): 2 bit level operations (to extract an entry in the word s) and one table lookup (to extract the actually stored point). So the total time complexity is equal to $4 + 3(k + 1)$ table lookups.

2. The two leaves belong to two different microtrees. Let k_1, k_2 be the number of points of the output that belong in these two microtrees and k_3 be the number of the remaining output points. Then the time bound is bounded from above by $4(k_1 + 1) + 4(k_2 + 1) + 2k_3$ table lookups, plus a constant number of table lookups and lowest common ancestor (*lca*) computations.

As a result, our approach is more complex, in its construction, but faster than the Cartesian Tree Approach. Also, as shown in the sequel, the storage requirements will be increased by a small constant factor.

Space Comparison

The first solution needs:

- M records for the table A
- N records for the table $L()$
- each node v of cartesian tree requires 4 records for each of the numbers $d(v)$,$h(v)$,$I(v)$ and A_v respectively plus the record that stores the point $p(v)$.
- Thus, the total space requirements of the first solution are $5N + M$ records.

Our solution needs:

- M records for each of the three tables A on each of the three layers.
- The first layer stores the points $p(v)$ on $N/\log N$ records. In addition, each node shares 2 path lists, that means we need $2(N/\log N)$ records additionally for each of the persistent lists P,L and R respectively.
- The second layer stores the points $p(v)$ on $(\log N/\log\log N)(N/\log N) = N/\log\log N$ records. We need $2(\log N/\log\log N)(N/\log N) = 2N/\log\log N$ records additionally for each of the persistent lists P,L and R respectively.
- The third layer stores the points $p(v)$ on $(\log\log N)(N/\log\log N) = N$ records.Also,the table lookup of each microtree needs 4 records,3 records for the 3 arguments and 1 record for the lookup result.
- Thus, the total space requirements of our solution are: $3M + 7(N/\log N) + 7(N/\log\log N) + N + 4N = 3M + 5N + 7N[(1/\log N) + (1/\log\log N)] \approx 3M + 5N$ records.

3.2.5 Experimental Comparison

We implemented both solutions in *Visual C++ 6.0* (see [17]) and we executed the two programs in Pentium based PC with the following hardware and software characteristics.

- Running at 700MHZ
- 128 MB of RAM
- 4 Gbyte partition hard disk space (total partition's space: 17 Gbytes)
- 2000 Server Operating System

Included in the $<$ *time.h* $>$ header, we used the *clock()* function that counts the number of executed cycles by the time passed after a user has given the query. We finally divided that number of cycles by $Clocks\_Per\_Sec$ of system in order to evaluate the *cpu time* in seconds (or msec). In order to evaluate a non zero number of cycles, we included this procedure within a while - loop of $10 \cdot Clocks\_Per\_Sec$. The final results are computed by dividing the cycle number by the number of loops.

First we ran our code for small input sizes f.e. $|S| = 16, 32, 256$ elements, in order to prove that even if the number of recursive *lca* routines in Cartesian tree is small, our solution is still better.

We choose for testing, particular input sets, where the minimum y-coordinate is always in the middle position of the whole set and the same goes on for the

Type of query	Cartesian Tree	Our Solution
$[1, 10] \times (-00, 8]$	0.000008 sec	0.000006 sec
$[1, 12] \times (-00, 11]$	0.000010 sec	0.000007 sec
$[1, 14] \times (-00, 5]$	0.000011 sec	0.000005 sec
$[1, 8] \times (-00, 10]$	0.000007 sec	0.000007 sec
$[1, 15] \times (-00, 15]$	0.000011 sec	0.000009 sec

Table 1: cpu times of each query

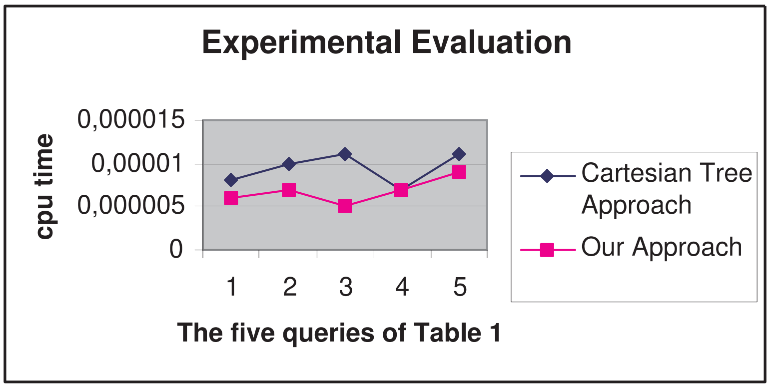

Figure 4: The respective graph of Table 1.

left and right subsets recursively. The reason is to construct always a balanced Cartesian tree, where the *lca* routine can be implemented simply. Concerning Cartesian Tree, this case of input sets is the "good" case. In contrast, if the input set is arbitrary then there is a high probability to construct an arbitrary non-balanced Cartesian Tree or a simple linear linked-list in worst-case. In these "bad" cases we must execute additional operations in order to transform the arbitrary Tree T to a balanced compressed tree C, and generally the whole *lca* computation is more complicated. Our solution is independent on input sets because is always balanced and the *lca* implementation is always simple.

It is obvious that if in these 'good' cases (for the Cartesian Tree) our solution has still better time performance the same will be occurred for any other case. Table 1 and the respective graph of Figure 4, show the previous results of the execution of five three-sided queries in a small set of 256 elements.

We continued by running several number of three-sided queries and depend on the size k of each answer we concluded in the following experimental graph in a set of 1000000 elements (see Figure 5).

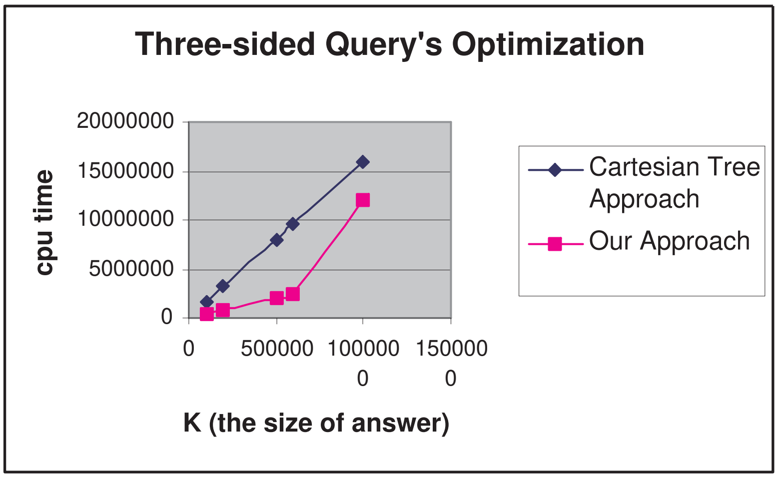

Figure 5: Cpu times depends on size k

As it can be seen from the experimental results our solution is better. We expected such a result because the query time of the first solution is approximately k times the *cpu time* for the execution of a lowest common ancestor query. Our solution executes a constant number of lca queries and in addition it uses sequential linear traversals of persistent lists and auxiliary arrays in order to retrieve the output points. The experimental results prove that in practice access of the positions of the lists and of the auxiliary arrays is executed rapidly and much faster than a set of lca queries. This can be explained from the fact that our solution clusters the reported points in consecutive memory locations and as a result the algorithm benefits from the use of fast multilevel caches in modern computers. With these multilevel caches one access of a record to disk or main memory results to the prefetching of sequential blocks of records in the memory hierarchy. Since access to cache is faster than access to main memory or disk, future access to the prefetched data is accelerated. Due to the same asymptotic behavior of the solutions above, we also expect that for a very large size of k (and consequently very large input size) the two approaches will have the same experimental behavior.

4 *D*-dimensional Range Searching

Let S be a set of N d-dimensional points, $d \geq 3$. The goal is to structure them in a data structure so that the points that lie in a query range of the form $[a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$ can be reported efficiently. We will show how to use our modified priority search tree, in order to solve the problem in $O(\log^{d-1} N)$ query time and $O(N \log^{d-2} N)$ space.

First we show how to solve the 3-dimensional problem for queries of the form $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$. We will normalize the coordinates of the points. To do so we will use two balanced binary trees each storing respectively the 1-, 2-coordinates of the points. In these trees we store for each coordinate its rank in the tree, which is an integer between 1 and N. Then we replace the coordinates of the points with their respective ranks. We call the new set of points the normalized set, and denote it by S'.

Given a query $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$ we will normalize it in the following way. We search for a_1 and b_1 in the tree storing the first coordinates of the points. Let p_1 be the largest coordinate, which is less than a_1 and let q_1 be the smallest coordinate greater than b_1. We denote by a'_1, b'_1 the ranks of p_1, q_1. Similarly we find the values a'_2, b'_2. This step takes $O(\log N)$ time. The points of S' that lie in $[a'_1, b'_1] \times [a'_2, b'_2] \times (-\infty, b_3]$ are the normalized versions of points of S that lie in $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$. So we need only solve the normalized problem on the grid $[1, M] \times [1, M] \times R$.

We construct a two-level structure. The first level is a balanced binary tree T that stores in its leaves the points of S, sorted according to their first coordinate. With each internal node u, we associate a secondary structure $D(u)$, for the points stored in the subtree of u. Each $D(u)$ is implemented as the structure of Theorem 7, for the last two coordinates of the points. For each such structure we will not use the array A. This is because each array needs $\Theta(M)$ space and all of them require $\Theta(M^2)$ space. Instead we will use the fractional cascading technique of Chazelle and Guibas [7]. With fractional cascading we can search for a specific value in a set of m secondary lists, containing a total of N points, in $O(\log N + m)$ time.

Now consider a query with the normalized range $[a'_1, b'_1] \times [a'_2, b'_2] \times (-\infty, b_3]$. We find $O(\log N)$ nodes in T such that they store the points with first coordinate in $(a'_1, b'_1]$. For each node u we query the structure $D(u)$ with the range $[a'_2, b'_2] \times (-\infty, b3]$. Through fractional cascading, the leaves on the search paths for the values a'_2, b'_2 can be found in $O(\log N + \log N) = O(\log N)$ time. So the query time is $O(\log N + t)$. The space is $O(N \log N)$ since each point is stored in $O(\log N)$ secondary structure and each secondary structure needs linear space. Therefore we have proved the following:

Lemma 8. *The three dimensional static range searching problem with half infinite ranges of the form* $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$ *can be solved in* $O(N \log N)$ *space and* $O(\log N + t)$ *query time.*

Now we consider the general case of arbitrary ranges $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$. As in [12] we use a technique of Edelsbrunner [9]. Let T be a balanced binary

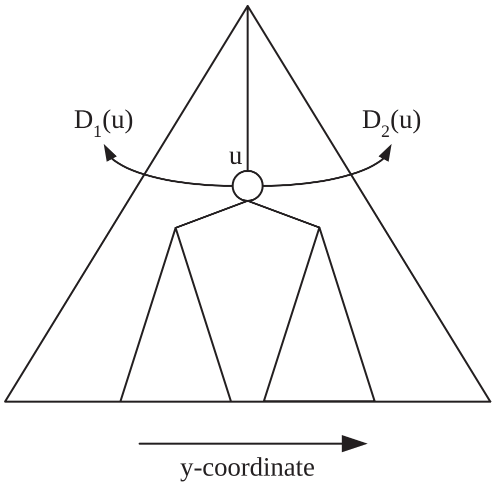

Figure 6: The D_1 structures answer queries with ranges of the form $[a_1, b_1] \times [a_2, b_2] \times [a_3, +\infty)$, while the D_2 structures answer queries of the form $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$.

tree which stores in its leaves the points according to their third coordinate. To each internal node u we associate two sets of points S_1, S_2. The set S_1 contains the points in the left subtree of u while the set S_2 contains the points in the right subtree of u. Each set S_1 is stored in a secondary structure D_1, while each set S_2 is stored in a secondary structure D_2, both implemented as the structure of Lemma 8 (see Figure 6). The D_1 structures answer queries with ranges of the form $[a_1, b_1] \times [a_2, b_2] \times [a_3, +\infty)$, while the D_2 structures answer queries of the form $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$. Since each point is stored in $O(\log N)$ secondary structures, the space is $O(N \log^2 N)$.

To answer a query with the range $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$, we find in T, the node u where the search paths for a_3 and b_3 split. Then we query the structure $D_1(u)$ with $[a_1, b_1] \times [a_2, b_2] \times [a_3, +\infty)$ and $D_2(u)$ with $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3]$. This takes time $O(\log N + t)$.

The correctness is proved as follows. Since the search paths for a_3 and b_3 split in u it follows that the points in $S_1(u)$ have third coordinate smaller than b_3 and the points in $S_2(u)$ have third coordinate greater than a_3. Furthermore, all points with third coordinate in $[a_3, b_3]$ are in $S_1(u) \bigcup S_2(u)$ so the query algorithm is correct. Therefore:

Theorem 9. *The three dimensional static range searching problem with iso-oriented rectangular ranges can be solved in $O(N \log^2 N)$ space and $O(\log N + t)$ query time.*

To solve the d-dimensional problem for $d > 3$ we use a structure with $d-2$ levels. The first level is a balanced binary tree, which stores in its leaves the points according to their first coordinate. To each node we associate the set of points stored in its subtree. This set is stored, in the same way, in a second level structure according to the second coordinates of the points. The construction continues in the same way, until the $(d-2)$-th level, which we implement with the structure of Theorem 9 for the last three coordinates of the points. It follows easily that each level incurs an increase by a $\log N$ factor to the query time and space of the structure, that is:

Theorem 10. *The d-dimensional static range searching problem on iso-oriented rectangular ranges can be solved in $O(N \log^{d-1} N)$ space and $O(\log^{d-2} N + t)$ query time.*

Remark. In [1] a new data structure was presented for range searching in three dimensions. This structure has query time $O(\log n + k)$ and needs $O(n \log^{1+\epsilon} n)$ space. It is also shown that the above bounds can be extended for any fixed d $(d \geq 4)$, with a penalty factor $O(\log^{d-3+\epsilon} n)$ in space and $O((\log n / \log\log n)^{d-3})$ in query time. Their solution, uses as a component, the structure of [12] for three-sided range queries. We can incorporate in their structure, instead of [12], our solution, thus getting a faster alternative with the same bounds.

The structure of Theorem 10, can be easily used to answer partial range queries, if we consider partial range queries as a special case of range queries where we are able to specify infinite ranges for some of the dimensions. In this case, we follow the same query approach as previously with the difference that when we face an infinite range in the i-th dimension of the query point, we proceed immediately to the secondary structure of the root of the respective i-th nested. Thus we get the following lemma:

Lemma 11. *The d-dimensional partial range problem, where the query has s specified and $d-s$ unspecified ranges, can be solved in $O(N \log^{d-1} N)$ space and $O(\log^{s} N + (d-s) + t)$ query time.*

Remark. If we know in advance that the unspecified ranges belong to the first $d-3$ levels, then the query time of Lemma 11 becomes $O(\log^{s-2} N + (d-s) + t)$.

5 Exact and Partial Match Geometric Retrieval

The problem of exact and partial match searching is one of the most extensively studied problems in the field of database systems. Lee and Wong in [16] have proposed the *Quintary tree*, a file structure for d-dimensional databases that achieves efficient times for all types of queries (exact match, partial match, range

	Point 1	Point 2	$\cdots$	Point N
A_1	A_{11}	A_{12}	$\cdots$	
A_2			$\cdots$	
$\vdots$				
A_λ	$A_{\lambda 1}$	$A_{\lambda 2}$	$\cdots$	$A_{\lambda N}$
$\vdots$				
A_d			$\cdots$	

Table 2: A d-dimensional geometric file of N points.

and partial range) at the expense of extra storage. The version of range tree described in [12], is a also a d-dimensional data structure that achieves $O(d \log N)$ time for exact match and $O(log^{d-2} N)$ for partial match queries (since it handles this query as a common range query). The required storage is $O(N \log^{d-1} N)$. The Fusion tree [11] supports the exact match searching, in one-dimensional space, in $O(\log N / \log\log N)$ query time, using linear space. The Exponential Search tree proposed by Andersson [2], improves even more the exact match searching by achieving $O(\sqrt{\log N})$ time using $O(N)$ storage. Our approach relies on the ideas and techniques used in the Quintary tree (which in essence is a preliminary version of a range tree) and the Exponential Search tree.

5.1 First Solution

Our approach uses a d-dimensional balanced binary tree as a skeleton structure along with d Exponential Search trees, one for each level, which are used as index structures in order to speed up the queries. The two parts of the structure are described in the following sections. We will describe these two parts for the general example of the table 2 below, in which there is at least one A_λ row $(1 \leq A_\lambda \leq d)$, that has N distinct values (f.e. in the table 2 below each element $A_{\lambda j}$ $1 \leq j \leq N$ of λ-th dimension is the superkey of the point (record), in which it belongs to)

We use as skeleton structure, the structure of Lemma 4. The index structures are d Exponential Search trees, one for each level of the skeleton structure (see Figure 7).

The i-th $(1 \leq i \leq d)$ index structure stores in its N leaves the nodes of the skeleton trees, of level i (i-th level balanced binary trees), each of which appears $\log^{i-1} N$ times, in the skeleton structure of level $i-1$. Thus from each of the N leaves of the i-th Exponential Search tree, begin $\log^{i-1} N$ pointers to the corresponding nodes of the i-th level balanced binary trees. The $\log^{i-1} N$

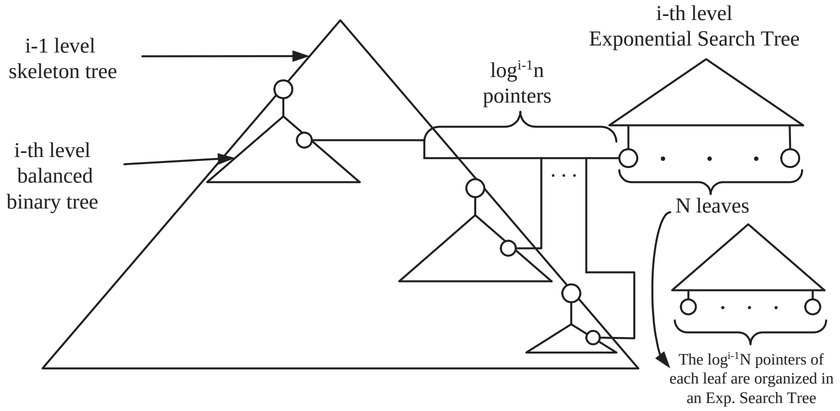

Figure 7: The i-th level of the skeleton structure along with its corresponding index.

pointers are organized in additional Exponential Search trees, N for each of the i index structures. For the above table we name the λ-th index structure as the primary tree. In each leaf node $a_{\lambda j}$ $(1 \leq j \leq N)$ of the primary tree we add a list which stores the tuple (record) that contains the $A_{\lambda j}$ superkey.

5.1.1 Worst Case Space and Time Complexity

The construction of our structure is achieved by executing a 2-step building process. The first step is the construction of the skeleton structure and the second the construction of the indexes for each level of the skeleton structure.

The algorithm presented in Figure 8 gets as input a file F of records and the dimension d of the associated file space. It returns a pointer to the root of the tree for the file F if $d \geq 1$. Otherwise it returns a pointer to a list of page numbers where the records of F reside.

Procedure BUILD_EXP_POINTERS(i) creates an exponential search tree for the pointers which begin from leaf i of $Exp\,[k]$ (we represent this additional tree as $Exp\,[k,i]$), that organizes efficiently the pointers from this leaf to its copies, on the nested balanced binary tree. In the worst case the leaf i appears in $log^{d-1}N$ subtrees, in the d-th level of the nested balanced binary tree. If j is the symmetric order of the copy subtree in which the leaf appears, the j-th leaf of the additional $Exp\,[k,i]$ tree is an index(pointer) to the node i of j-th copy subtree.

The total space required for the structure is the space of the skeleton tree

Procedure Build(d,f)
1. Create the structure of Lemma 4
2. **for** $k = 1$ **to** d **do**
3. BUILD_EXP(k)
4. **for** $i = 1$ **to** N **do**
5. BUILD_EXP_POINTERS(i)

Figure 8: Procedure BUILD_EXP creates the exponential search tree for dimension k.

plus the space of the indexes along with the additional Exponential trees used to organize the pointers to the skeleton structure.

Theorem 12. *The space required for the whole structure is* $O(N \log^{d-1} N)$.

Proof. The skeleton structure requires $O(N \log^{d-1} N)$ space (see lemma 4). Since the additional indexes occupy space proportional to the number of nodes in the skeleton structure, the theorem follows immediately.

The query Q is assumed to be a vector of d-tuples $(x_1, x_2, \ldots, x_d)$. In the case of a partial match query x_i may be an $*$, if the i-th query key is unspecified.

Procedure Exact_Match($Record\,[x_1, x_2, \ldots, x_d]$)
1. Find_Index(x_λ)
2. Retrieve the record stored in the node reported from step 1.

Figure 9: The procedure Find_Index finds x at the index of level i (or primary tree $Exp\,[i]$).

Theorem 13. *The exact match query requires* $O(\sqrt{\log N} + t)$ *time in the worst case.*

Proof. Step 1 needs $O(\sqrt{\log N})$ time in order to search for x_i in the primary tree.

Step 2 requires $O(t)$ time for retrieving the record $[x_1, \ldots, x_d]$.

Thus the total time is: $O(\sqrt{\log N} + t)$ (in this case $t = d$)

Theorem 14. *The partial match query requires* $O(t + (d + s\sqrt{\log N}))$ *time in the worst case, where* t *is the number of records retrieved and* s *is the number of keys specified.*

Procedure Partial_Match($Record\,[x_1, x_2, \ldots, x_d]$)
1. **for** $i = 1$ **to** d **do**
2. **If** $x_i = *$ (unspecified) then access the secondary structure stored at the root of the respective subtree
3. If $x_i \neq *$ (specified) then perform an exact match search for x_i and access the node pointed to by the index structure ($Exp\,[i]$)
4. Retrieve all records which are included in the tree which has as its root the current node (the node returned by step 1 or 2).

Figure 10: The partial match algorithm.

Proof. The first step is executed $d - s$ times and requires $O(1)$ each time.

The second step is executed s times and requires $O(\sqrt{\log N} + \sqrt{\log\log^{d-1} N})$ each time.

Step 3 requires $O(t)$ time.

Thus, if t is the number of retrieved records, then the total query time becomes $O(t + ((d - s) + s(\sqrt{\log N} + \sqrt{\log\log^{d-1} N}))) = O(t + (d + s(\sqrt{\log N} + \sqrt{(d-1)\log\log N - 1})) = O(t + (d + s\sqrt{\log N}))$.

6 The Second Solution[1]

The main idea behind our second approach is to extend the Exponential Search tree in order to support efficient exact match and partial match queries in d-dimensional space, where $d \geq 2$. Our solution combines a nesting structure of Exponential search trees along with additional information stored at each root node in the form of balanced binary trees. The Exponential Search trees are used for answering exact match queries while the additional information in each root node is used for answering partial match queries. As in the case of Quintary tree, our structure stores some tuples more than once resulting in a slight increase in the amount of required storage space. The new data structure is a set of nested Exponential Search trees (the skeleton structure), one for each dimension d. Since Exponential Search trees store information in their leaf nodes, each leaf node at level i, is the root of a new tree at level $i + 1$. This nesting structure stores information needed mostly for exact match queries.

At the root of each tree of level i there is an additional nested exponential Search tree ($\forall$_*subtree*) for $d - i$ levels which stores in its leaves all the information of level i. These Exponential Search trees are used for answering partial

[1] A preliminary version of this solution has been presented in [22].

match queries, which have the key for i-th level unspecified. Although our solution uses additional space, since we store some keys more than once, we achieve faster query responses for d-dimensional space when $d \geq 2$.

6.1 The Build Process

The construction of our structure is achieved by executing a 3-step building process. The first step is an *top-down* building process, which diffuses the information from the root of the tree to its leaves and constructs the skeleton structure. The other step is a *bottom-up* building process, which adds the middle subtree in the proper nodes and diffuses the information from the last level to the root. The third step uses superkey information to add pointers that are used to speed up exact match queries.

We will describe these three steps for the general example of the table 2 we described earlier in which the A_λ row has N distinct values (that is A_λ contains the superkey of the geometric file).

Step1: The top-down building process

Firstly, we construct an Exponential Search tree using the elements of the first row A_1. We represent this tree as $Exp\,[1]$. According to the definition of the Exponential Search tree, the elements of the first row $A_{11}, \ldots, A_{1N}$ (the distinct number of which is D_1) are stored in the leaves. For the D_1 leaves of $Exp\,[1]$ we construct pointers to the roots of the D_1 Exponential Search trees of level 2 ($Exp\,[2]$) respectively. Each of the $Exp\,[2]$ trees stores in its leaves the elements of the second row A_2. At level 2 there are $D_1 \cdot D_2$ leaves.

The remaining $d-2$ levels are constructed in the same way: for the construction of j-th level ($j \in [1, d]$), from each of the $D_1 \cdot D_2 \cdots D_{j-1}$ leaves of the previous level ($(j-1)$-th level) there are $D_1 \cdot D_2 \cdots D_{j-1}$ pointers to the roots of the $Exp\,[j]$ trees respectively each of which stores the j row of table.

Step 2: the bottom-up building process

The building process begins at the $d-1$ level. In each root node $u_{d-1,i}$ of the i ($1 \leq i \leq D_1 \cdot D_2 \cdots D_{d-2}$) Exponential Search trees ($Exp\,[d-1]$), we add a middle subtree. This subtree is a new Exponential Search tree, which contains all information stored in the Exponential Search tree of level d ($Exp\,[d]$). We represent the $T(u_{d-1,i})$ subtree (that is the subtree which has as its root the node $u_{d-1,i}$) as $\forall Exp\,[d-1]\,[d]$. At each root node of level $d-2$ we add as a middle subtree the $\forall Exp\,[d-1]\,[d]$. We represent the $T(u_{d-2,i})$ subtree as $\forall Exp\,[d-2]\,[d-1]\,[d]$.

The process continuous until we reach the top level of the skeleton structure.

Step 3: Superkey pointer addition

As we described, the A_λ row of the table contains the superkeys of the geometric file. In each leaf node $a_{\lambda j}$ $(1 \leq j \leq N)$ of one of the $Exp\,[i]$ trees, we add a list which stores the tuple (record) that contains the $A_{\lambda j}$ superkey. We name this tree the *Record_Tree* of the structure. Next we make a node with two pointers, the partial_field and the Exact_field. We name this node the *Troot node*. The first pointer points to the root of the $Exp\,[1]$ tree and the other to the root of the Record_Tree.

6.2 Time and Space Analysis

In this subsection we present the algorithms for exact and partial match queries and we analyze the time and space complexity of our structure.

Procedure Exact_Match($Record\,[x_1, x_2, \ldots, x_d]$)
1. Search for x_λ in the Record_tree
2. Retrieve the record $[x_1, x_2, \ldots, x_d]$ that is pointed by the returned leaf.

Figure 11: The exact match algorithm.

Theorem 15. *The exact match query requires* $O(\sqrt{\log N} + t)$ *time.*

Proof. We need $O(\sqrt{\log N})$ time to search for x_λ in the Record_tree and $O(t)$ time for retrieving the record $[x_1, \ldots, x_d]$ which is pointed by the returned leaf. So, the total time is $O(\sqrt{\log N} + t)$ (in this case $t = d$).

Procedure Partial_Match($Record\,[x_1, x_2, \ldots, x_d]$)
1. **for** $i = 1$ **to** d **do**
2. **if** x_i is unspecified **then**
3. In the root of subtree $Subtree\,[Root\,[Exp\,[i]]]$ search in the middle-tree
4. **else**
5. Search the leaf x_i in the skeleton $Exp\,[i]$ tree.

Figure 12: The partial match algorithm.

Theorem 16. *The partial match searching takes* $O(t + (d + s\sqrt{\log N}))$ *time.*

Proof. The **if** statement is executed $d-s$ times and the **else** statement is executed s times. The **else** statement takes $O(\sqrt{\log N})$ in each of the s loops. So, if t is the number of retrieved records then the total time becomes: $O(t + ((d-s) + s\sqrt{\log N})) = O(t + (d + s(\sqrt{\log N} - 1)) = O(t + (d + s\sqrt{\log N}))$.

Remark. The $O(\sqrt{\log N} + t)$ and $O(t + (d + s\sqrt{\log N}))$ query times of Theorems 9,12 and 10,13 respectively, can be reduced to $O((\sqrt{\frac{\log N}{\log\log N}} + t)$ and $O(t + (d + s\sqrt{\frac{\log N}{\log\log N}}))$ respectively by using instead of the Exponential Search Tree, the linear space data structures proposed for the predecessor problem in [3, 5].

Theorem 17. *The structure requires $O(N \cdot C^{d-1})$ space, where $C = \frac{\sum_{i=1,i\neq j}^{d} D_j}{d-1}$.*

Proof. The Exponential search tree requires linear space. Our d-dimensional structure requires in average case $O(N \cdot C^{d-1})$ storage, where C is the average number of $D_j (j \in [1, d] \, \text{and} j \neq i)$ of all rows except the i-th row, which stores the superkeys of the table and obviously has N distinct values (or keys). So, $C = \frac{\sum_{i=1,i\neq j}^{d} D_j}{d-1}$.

Contrary to the Range and Quintary trees, which require $O(N \log^{d-1} N)$ and $O(\frac{N \log^d N}{(d-1)!})$ space respectively, our solution requires more space if $C > \log N$ and less space if $C < \log N$, which is a usual case.

7 Conclusions

In this paper we present new multidimensional searching algorithms that formulate both theoretical and practical superiority using the RAM model with word size w. The first solution of exact and partial match searching uses as a skeleton structure our modified (in last three levels) range tree, we described first. So, we have developed an integrated data structure that supports rapidly all four types of queries (range, partial range, exact match and partial match queries).

On the other hand, the second solution is more concrete, in the sense that no additional index structures are used while it can handle efficiently exact and partial match searching queries. The above algorithms could be used in many application areas such us computer graphics, computer vision, database management systems, computer-aided design, solid modeling, robotics, geographic information systems (GIS), image processing, computational geometry, pattern recognition, and other areas. On database management systems, it could be very important the implementation of these structures in secondary memory or in cache oblivious model. Finally, an open problem is the elimination of persistence amongst the lists in order to simplify the construction and improve the storage requirements of our structure. Probably, a more clever partitioning of the modified priority search tree T will be a good start.

References

1. S. Alstrup, G.S. Brodal, T. Rauche, New Data Structures for Orthogonal Range Searching. In Proc. 41st Annual Symposium on Foundations of Computer Science, pages 198-207, 2000
2. A. Andersson, Faster deterministic sorting and searching in linear space. In 37th Annual Symposium on Foundations of Computer Science, pages 135-141, Burlington, Vermont, 14-16 October 1996.
3. A. Andersson and M. Thorup, Tight(er) worst case bounds on dynamic searching and priority queues, In ACM Symposium on Theory of Computing (STOC), 2000.
4. P. K. Agarwal, Range searching, Technical Report CS-1996-05, Dept. of Computer Science, Duke University, 1996.
5. P. Beame and F. Fich, Optimal bounds for the predecessor problem. In 31st ACM Symposium on Theory of Computing (STOC), 1999.
6. B. Chazelle, Filtering search A new approach to query answering, SIAM J. Comput.15 (1986), pp. 703-724.
7. B. Chazelle and L. Guibas, Fractional cascading: I, A data structuring technique; II, Applications, Algorithmica 1 (1986), pp.133-191.
8. J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan, Making data structures persistent, J. Comp. Syst. Sci. 38 (1989) pp. 86-124.
9. H. Edelsbrunner, A note on dynamic range searching, Bull. EATCS 13 (1981), pp. 34-40.
10. O. Fries, K. Mehlhorn, S. Naher and A. Tsakalidis, A loglogn data structure for three sided range queries, Inform. Process. Lett. 25 (1987), pp. 269-273.
11. M. L. Fredman and D. E. Willard, Surpassing the information theoretic bound with fusion trees, J. Computer Systems Science 47, pp: 424-436, 1994.
12. H. N. Gabow, J. L. Bentley, and R. E. Tarjan, Scaling and related techniques for geometry problems, in Proceedings, l6th Annual ACM Symp. on Theory of Computing, 1984, pp. 135-143.
13. H. N. Gabow and R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, in J. Comput. Syst. Sci. 30(1985) 209-221
14. D. Gusfield, Algorithms on Strings, Trees and Sequences, Computer Science and Computational Biology, Cambridge University Press,1994.
15. D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestor, SIAM J. Comput.13 (1984), pp. 338-355.
16. D. T. Lee and C. K. Wong, Quintary Trees : A File Structure for Multidimensional Database Systems, ACM Trans. on Database Systems, Vol 5, No 3, pp: 339?353, 1980.
17. Ch. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, J. Tsaknakis, B. Vassiliadis, Geometric Retrieval for Grid Points in the RAM Model, Technical Report in Computer Technology Institute, CTI TR2002/11/02.
18. E. M. McCreight, Priority search trees, SIAM J. Comput. 14 (1985), pp. 257-276.
19. K. Mehlhorn, Data Structures and Algorithms. Multidimensional Searching and Computational Geometry, Springer Verlag, 1984.
20. M. H. Overmars, Efficient data structures for range searching on a grid, J. Algorithms 9 (1988), pp. 254-275.
21. B. Schieber and U. Vishkin, On finding lowest common ancestors: simplifications and parallelization, SIAM J. Comput., 17:1253-62, 1988.
22. S. Sioutas, A. Tsakalidis, J. Tsaknakis, V. Vassiliadis, An Efficient indexing on Database Structure for Exact and Partial Match Queries, Moscow ACM SIGMOD chapter, ADBIS'99, Maribor, Slovenia, September 13-16, 1999.
23. J. Vullemin, A unifying look at data structures, C. ACM 23 (1980) pp. 229-239.
24. P. van Emde Boas, Preserving Order in a forest in less than logarithmic time and linear space, Information Processing Letters 6(3), 80-82, 1977.