Partitioning range trees

Mark H. Overmars and Michiel H.M. Smid

RUU-CS-87-3
February 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Partitioning range trees

Mark H. Overmars and Michiel H.M. Smid

Technical Report RUU-CS-87-3
February 1987

Department of Computer Science
University of Utrecht
3508 TA Utrecht
the Netherlands

PARTITIONING RANGE TREES

by

Mark H. Overmars
Department of Computer Science
University of Utrecht

and

Michiel H.M. Smid
Department of Computer Science
University of Amsterdam

abstract : In this paper we will describe a number of methods to partition range trees, in order to be able to maintain them efficiently in secondary memory. We show that we can partition a range tree, such that after an update only O(log*n) parts, each of size O(n) will have changed. Also, we give a partition, such that after an update O(log n) parts, each of size only $O(\sqrt{n})$ will change.This shows that it is indeed possible to make interesting non-trivial partitions of geometric data structures and, hence, provide interesting directions for the research on shadow administrations.

February 1987

1. Introduction

1.1. The reconstruction problem for dynamic data structures

A substantial part of the research in the theory of data structures is concerned with the design of structures and algorithms solving searching problems. In a searching problem, a question (query) is asked about an object x with respect to a given set S of objects. An example is the orthogonal range searching problem.

Definition 1 : Let S be a set of points in d-dimensional space, and let $([x_1:y_1],[x_2:y_2],...,[x_d:y_d])$ be some hyperrectangle. The orthogonal range searching problem asks for all points $p=(p_1,p_2,...,p_d)$ in S, such that $x_1 \leq p_1 \leq y_1$, $x_2 \leq p_2 \leq y_2$, ..., $x_d \leq p_d \leq y_d$.

The range searching problem frequently occurs in the theory of data bases. Consider e.g. a salary administration in which the information for each registered person includes age and salary. We can view each person as a point in 2-dimensional space, with as first coordinate the age, and as second coordinate the salary. Then a question like "give all persons with age between 20 and 25, having a salary between $30,000 and $35,000 a year", is an example of a range query.

A solution to a searching problem consists of a data structure, representing the set S, together with an algorithm which answers queries efficiently.
Often, the efficiency of such data structures is caused by the facts that they can be stored entirely in main memory, and that they are dynamic (i.e., they can be maintained efficiently if points are inserted or deleted in the set S).
However, if we take the imperfection of computer systems into account, both nice properties pass away. After a system crash, or as a result of errors in software, the contents of the main memory can get lost. Another case, in which the data structure in main memory can get lost, is the regular termination of the application program which uses the structure. In case of an application which is executed on a system which is also used by other persons, the copy of the data structure in core will get lost between two runs of the application program.
In both cases, the data structure has to be reconstructed from the information stored in secondary memory (this information is called the shadow administration).
One possibility to solve this problem is to keep in secondary memory a copy of the data structure. Then after a crash, the only thing we have to do is to transport the copy to core.
However, the main problem in this case is to maintain the copy efficiently after an update of the data structure in core.
We can, of course, after an update transport a new copy of the entire data structure to secondary memory. Suppose, e.g., that our data structure is a balanced binary tree, representing a set of n elements. Then we have to transport after each update an amount of O(n) data, whereas actually only O(log n) of it has changed (in this paper all logarithms are to base 2).

Therefore we might try to partition the data structure, such that after an update only a small number of parts and, hence, only a small amount of data has to be transported.

The partitioning of data structures also has the following important application.
Suppose we have a data structure which is too large to fit in main memory and, hence, must be stored in secondary memory. Then, in order to answer queries or to perform updates, parts of the data structure have to be transported from secondary memory to core, and vice versa.
Clearly, also in this application it is useful to partition the structure, such that only a small amount of data has to be transported.
However, in this paper, this application will not be considered any further (see the final version [5] for more details of this application). All results will be given in terms of shadow administrations.

In order to be able to analyze the efficiency of partitions, we have to make some assumptions how secondary memory is organized.
We assume in this paper that the file in secondary memory is divided into blocks of some fixed size. These blocks are linked by pointers: Block i contains a pointer to the physical address of block i+1. There is the ability of direct block access: It is possible to access a block directly, provided its physical address is known. Furthermore it is possible to replace a block by another block, or a number of (physically) successive blocks by at most the same number of blocks. Finally, a new block, or a number of successive new blocks, can be added at the end of the file.
Now suppose we have partitioned our data structure into a number of parts. Then we store a copy of the data structure in secondary memory by putting each part of the partition in a number of (physically) successive blocks.
In core, we record for each part of the partition the address of the beginning of its copy in secondary memory.
After an update of the data structure in core, the copy is updated by replacing all parts of the partition that actually have changed.

We express the complexity of this update procedure by:
(i) The number of seeks that has to be done: If we transport a number of successive blocks to secondary memory, then we have to do one seek. Hence after an update the number of seeks to be done is equal to the number of parts of the partition that have changed (and thus have to be replaced).
(ii) The total amount of memory that has to be transported from core to secondary memory.

Note that if two parts of the partition are stored in successive blocks, these two parts can be replaced requiring only one seek. That is, the number of seeks necessary after an update depends on the way the parts are stored in secondary memory. However, since we do not think we can get essential gain by considering this storing problem, we assume here that the number of seeks after an update is equal to the number of parts that have changed.

The purpose of this paper is to show that it is possible to make non-trivial partitions of geometric data structures (in particular range trees), and, hence, that it is possible to maintain efficiently a data structure in secondary memory, with only a small number of seeks per update. To this end we consider a data structure for range searching by, e.g., Lueker [3] and design efficient partition schemes. The methods presented also work for a number of other data structures.

The paper is organized as follows.
In the next section we define the partitioning problem more precisely.
In Section 1.3 we define the basic concepts needed for the rest of the paper, namely BB[α]-trees and range trees.
In Chapter 2 we give efficient partitions of 2-dimensional range trees. In fact, we have to modify the definition of range trees somewhat, by requiring an extra balance condition.
In Chapter 3 we briefly describe how the results of Chapter 2 can be generalized to multi-dimensional range trees.
In Chapter 4 we consider storage management for the data structure in secondary memory.
Finally, in Chapter 5, we give some concluding remarks, and we indicate how the results of this paper can be extended to other geometric data structures.

1.2. The partitioning problem

The partitioning problem can be described as follows.
Let DS be a dynamic data structure. We want to partition DS into a number of parts, such that:
(i) Each part has about the same size.
(ii) Each update changes at most a small number of parts.
We define this concept more precisely.

Definition 2 : A partition of a dynamic data structure, representing a set of n elements, is called an (f(n),g(n))-partition, if
(i) Each part has size O(f(n)).
(ii) There are O(S(n)/f(n)) parts, where S(n) is the amount of space required to store the data structure.
(iii) An update changes O(g(n)) parts.

Note that it follows from (i), that if parts do not overlap, the number of parts is Ω(S(n)/f(n)).
In most cases, we will only be able to prove that an update changes O(g(n)) parts on the average.

The relation of this definition to the preceding section should be clear.
It states that in order to maintain a copy of the data structure in secondary memory, after each update we have to transport O(g(n)) parts, each of size O(f(n)). In other words, an update will take O(g(n)) seeks, and an amount of O(f(n)g(n)) data transport.

1.3. Preliminary results

In this section we define some basic concepts which we shall need in the rest of the paper.

Definition 3 : Let α be a real number, $2/11<\alpha\leq 1-1/2\sqrt{2}$.

A binary tree is called a BB[α]-tree, if for each internal node v, the number of leaves in the left subtree of v divided by the total number of leaves below v lies in between α and $1-\alpha$.

Obviously, in a BB[α]-tree a similar balance condition holds for the right subtree of each internal node.
In this paper, BB[α]-trees are used as leaf search trees. That is, if we want to use a BB[α]-tree T to represent a set S of real numbers, then we store the elements of S in sorted order in the leaves of T. Internal nodes contain information to guide searches in the tree. The following theorem gives the complexity of a BB[α]-tree (the proof can be found in Blum and Mehlhorn [2]).

Theorem 1 : Suppose the set S contains n elements. Then a BB[α]-tree for S requires O(n) space, and can be built in O(n log n) time. Furthermore, insertions and deletions can be performed in O(log n) time, by means of single and double rotations.

BB[α]-trees are the building blocks of range trees, which we will define now (cf. Bentley [1], Lueker [3], Willard and Lueker [9]).

Definition 4 : Let S be a subset of the d-dimensional real vector space.

A d-dimensional range tree T, representing the set S, is defined as follows.

If d=1, then T is a BB[α]-tree containing the elements of S in sorted order in its leaves.

If d>1, then T consists of a BB[α]-tree, called the main tree, which contains in its leaves the elements of S, ordered according to their first coordinates. Furthermore, each internal node v of this main tree contains an associated structure, which is a (d-1)-dimensional range tree for those elements of S which are in the subtree rooted at v, taking only the second to d-th coordinate into account.

E.g., a 2-dimensional range tree for set S consists of a BB[α]-tree, containing in its leaves the points of S ordered according to their x-coordinates. Let v be an internal node of this tree, and let S_v be the subset of S represented by v. Then node v contains a BB[α]-tree, representing the set S_v, ordered according to their y-coordinates.
Later in this paper we shall modify the definition of range trees, in order to be able to maintain them efficiently in secondary memory.

Range trees are used to solve the orthogonal range searching problem (cf. Section 1.1).

Theorem 2 : Let S be a set of n points in d-dimensional space.

Then a d-dimensional range tree for set S can be built in $O(n \log^{d-1} n)$ time, and it requires $O(n \log^{d-1} n)$ space.

Using this tree, orthogonal range queries can be solved in time $O(\log^d n + t)$, where t is the number of reported answers

Finally, insertions and deletions can be performed in time $O(\log^d n)$, on the average.

Proof : The bounds for the building time, the storage requirement, and the query time are well known, and can be found, e.g., in Bentley [1].

After an update of the set S, the range tree can be maintained using the partial rebuilding technique (cf. Lueker [3], Overmars [4]) :

Suppose we want to insert or delete point p in a range tree T. Then we search with p in the main tree of T to locate its position among the leaves, and we insert or delete p in all the associated structures we encounter on the search path (if these associated structures are one-dimensional range trees, then we apply the usual insertion/deletion algorithm for BB[α]-trees; otherwise we use the same procedure recursively). Next, we insert or delete p among the leaves of the main tree, and we walk back to the root. During this walk, we locate the highest internal node v which has become out of balance as a result of the update. Then, we rebalance at node v by rebuilding the entire structure rooted at v as a perfectly balanced range tree (perfectly balanced means that for each internal node, the number of leaves in the left resp. right subtree differ by at most one).

Note that if node v is the root of the main tree, we have to rebuild the entire range tree, which takes time $O(n \log^{d-1} n)$. However, in this case $\Omega(n)$ updates must occur until we have again to rebuild the entire structure.In fact, Lueker [3] has shown the following:

Let v be a node in a range tree which is in perfect balance. Let n_v be the number of leaves in the subtree rooted at v, at the moment v gets out of balance. Then there must have been $\Omega(n_v)$ updates in the subtree of v.

Using this result, it can be shown that the above sketched insertion/deletion algorithm can be performed in time $O(\log^d n)$ on the average (cf. Lueker [3], Overmars [4]).

❑

In fact, insertions and deletions can even be performed in worst case time $O(\log^d n)$. See Willard and Lueker [9] for details.

2. Two-dimensional range trees

In this chapter we study partitions of 2-dimensional range trees. In order to give efficient partitions, we have to modify range trees somewhat.
First we give some trivial results.

2.1. Some trivial partitions

Theorem 3 : For a 2-dimensional range tree, there exists an (n log n,1)-partition.

Proof : Just use the entire tree as one part. ❑

Theorem 4 : For a 2-dimensional range tree, there exists a $(1,\log^2 n)$-partition.

Proof : Each node (either of the main tree, or of an associated structure) forms a part in its own. Since an update takes $O(\log^2 n)$ on the average, it changes $O(\log^2 n)$ parts on the average. ❑

Theorem 5 : For a 2-dimensional range tree, there exists an (n,log n)-partition.

Proof : Each level of the main tree, together with its associated information, forms a part. The bounds follow easily.

❑

Clearly Theorem 5 does not help anything and, in fact, is even worse than Theorem 3: We still have to transport an amount of O(n log n) data to secondary memory, and this requires O(log n) seeks rather than 1.

2.2. Modified range trees

In this section we try to divide the main tree into parts, whereas associated structures are never subdivided. This, of course, means that parts will have size $\Omega(n)$, because the tree associated with the root of the main tree has size $\theta(n)$.

First we want to improve the (n,log n)-partition of Theorem 5, to get an (n,log log n)-partition. The idea is as follows.
Suppose that the range tree is perfectly balanced, i.e., for each internal node, the number of leaves in its left resp. right subtree differ by at most one.
Now cut the main tree at level log log n.
Each level up to level log log n forms a part. This gives us O(log log n) parts, each of size O(n).
Each subtree which has its root at level log log n, represents O(n/log n) points. So it has size

$O((n/\log n)\log(n/\log n)) = O(n)$ and, hence, it can form a part. This gives us $O(\log n)$ parts, each of size $O(n)$.
So in total we have $O(\log n)$ parts, each of size $O(n)$, provided the tree is perfectly balanced.
However, as soon as we insert or delete points, the tree is not perfectly balanced anymore. In fact, the number of points represented by a subtree having its root at level $\log\log n$ can become $\Omega((1-\alpha)^{\log\log n} n) = \Omega((1/2\sqrt{2})^{\log\log n} n) = \Omega(n/\sqrt{\log n})$. Hence such a subtree may have size $\Omega(n\sqrt{\log n})$, which is too large to form a part.
(Of course, we can cut the main tree at a lower level, i.e., a level $\geq \log\log n$. However, then the number of subtrees having their root at this level (and hence the number of parts) becomes too large.)

In order to avoid that subtrees, having their root at level $\log\log n$, become too big, we modify the definition of range trees.

Let S be a subset of the 2-dimensional real vector space.
We suppose that the points of $S=\{p_1 \leq p_2 \leq p_3 \leq ... \leq p_n\}$ are ordered according to their x-coordinates.
Partition S into sets $S_1=\{p_1,p_2,...,p_{h(n)}\}$, $S_2=\{p_{h(n)+1},...,p_{2h(n)}\}$,..., where $h(n)=\lceil n/\log n \rceil$.

Definition 5 : A <u>modified range tree</u>, representing the set S, is defined as follows.

(i) Each set S_i is represented by an ordinary 2-dimensional range tree T_i.
Let r_i be the root of T_i. These roots are ordered according to $r_1 < r_2 < r_3 < ...$

(ii) The roots r_i are stored in a perfectly balanced augmented BB[α] leaf search tree T.
Let v be an internal node of T, representing the roots $r_i, r_{i+1}, ..., r_j$.
Then v contains an associated structure, which is an ordinary 1-dimensional range tree, representing the set $S_i \cup S_{i+1} \cup ... \cup S_j$, ordered according to their y-coordinates.

Theorem 6 : A modified range tree, representing a set of n points, requires $O(n \log n)$ space to store, and can be built in $O(n \log n)$ time.
Range queries can be solved, using this tree, in time $O(\log^2 n + t)$, where t is the number of reported answers.
Insertions and deletions in this tree can be performed in time $O(\log^2 n)$, on the average.

Proof : The bounds for the storage requirement, the building time and the query time can be proved in the same way as in Theorem 2.
An insertion or deletion of a point p is performed as follows.
First we walk down tree T, to find the appropriate root r_i. During this walk we insert or delete p in all associated structures we encounter on the search path. Then we insert or delete p in T_i, using the ordinary update algorithm for range trees (cf. Section 1.3).
Obviously, this procedure takes time $O(\log^2 n)$ on the average, provided each set S_i contains $\theta(n/\log n)$ points.
Suppose at the moment we build this structure, the set S contains n points. Then each set S_i

contains $\lceil n/\log n \rceil$ points.
As soon as at least one set S_i contains either $1/2\lceil n/\log n \rceil$ or $2\lceil n/\log n \rceil$ points, then we rebuild the entire data structure. That is, we partition the set S into subsets of size $\lceil m/\log m \rceil$, where m is the cardinality of S at that moment.
So every $\Omega(n/\log n)$ updates, we have to rebuild the data structure at most once.
It follows that the average update time of the modified range tree is $O(\log^2 n)$.

❑

Hence the modified range tree has (asymptotically) the same complexity as the ordinary range tree. Observe that, if we do not have to rebuild the structure, only associated structures of T are changed after an update. So there are no rotations to be done in T (after a rotation, certain nodes will move from one part to another, so this causes some trouble in maintaining the partition).

Theorem 7 : For a modified range tree, there exists an (n,log log n)-partition.

Proof : Each tree T_i has size O(n) and, hence, can form a part. This gives us O(log n) parts, each of size O(n).
Each level of the tree T, together with its associated structures, forms a part. Since tree T has depth O(log log n), this gives us O(log log n) parts, again each of size O(n).
Clearly, an update changes O(log log n) parts if we do not have to rebuild the data structure.
If we have to rebuild the structure, then O(log n) parts will change. Since this has to be done at most once every $\Omega(n/\log n)$ updates, the average number of parts that will change after an update is O(log log n).

❑

Remark : If we use the above (n,log log n)-partition, then after an update we have to transport O(n log log n) data to secondary memory, requiring O(log log n) seeks. If we allow parts to have varying size, then we can let T be a part in its own, of size O(n log log n). Then an update changes only two parts, namely the tree T and one tree T_i. So we have to transport again O(n log log n) data, but now this takes only 2 seeks.
In fact, we can even obtain this result with equal sized parts, as the following theorem shows.

Theorem 8 : For a modified range tree, there exists an (n log log n,2)-partition.

Proof : The tree T forms a part in its own. Furthermore, we combine log log n trees T_i into one part.

❑

The next theorem improves Theorem 7 considerably. Before we can state it, we have to define the function log*n. Let $\log^{(i)}n$ denote the i-th iterate of the function log n.
Then $\log^*n=\min\{i\geq 0 \mid \log^{(i)}n\leq 1\}$.The reader should observe that, although log*n goes to infinity as n does, for all practical values of n we have $\log^*n\leq 4$.

Theorem 9 : For a modified range tree, there exists an (n,log*n)-partition.

Proof : Since we want to partition the data structure into parts of size O(n), each tree T_i can form a part. This gives us O(log n) parts.
So we are left with the tree T. We first sketch how this tree is partitioned.
The root of T, together with its associated structure, forms a part. This removes the top level of T. Now consider the two sons v and w of the root. Look at the subtree consisting of v and its two sons. It takes, together with its associated structures, O(n) storage and, hence, can form a part. Similar for w. This removes two more levels of T, so we are left with 8 sons. For each of these sons u, we make a part consisting of the tree with root u and depth 8. This subtree, with associated structures, again uses O(n) space and, hence, can be a part. We now have removed 11 levels. So we are left with 2^{11} sons. For each son we take a subtree of depth 2^{11}, with associated structures, which takes O(n) storage. Next we are left with 2^k sons, where $k=2^{11}+11$, etc.
The reader should note, that the tree T is perfectly balanced, and that it remains perfectly balanced. So a node at level i indeed represents $\theta(n/2^i)$ points (cf. the discussion at the beginning of this section).
We will describe the above partition more precisely.
Let a(0)=0, $a(k+1)=2^{a(k)}+a(k)$ if $k\geq 0$. Let d be the depth of tree T (d is the number of nodes in the longest path in T). Since T is perfectly balanced, we have d=log log n + O(1).
Let $m=\min\{i\geq 0 \mid a(i)>d\}$. Then it is easy to show that m=log*n + O(1).
Now tree T is partitioned as follows.
For each k, $0\leq k\leq m-1$, there are $2^{a(k)}$ parts, where each part is a subtree of T, together with associated structures, with its root at level a(k), of depth $2^{a(k)}$.
Clearly, each part has size O(n).
Furthermore, the number of parts into which T is partitioned is

$$\Sigma_{k=0,\ldots,m-1}\, 2^{a(k)} = O(2^{a(m-1)}) = O(2^d) = O(2^{\log\log n + O(1)}) = O(\log n).$$

Finally, each update changes m=O(log*n) parts of T, if we do not have to rebuild the data structure. If we take the cost of rebuilding into account, then we see that on the average O(log*n) parts of the tree T will change.
As remarked before, only associated structures of T are changed after an update. So there are no rotations to be done in T.

❑

This is an interesting result, because it means that we can maintain a modified range tree in secondary memory, by transporting O(log*n) parts of size O(n) with every update, and in practical situations log*n can be considered a small constant.

2.3. Partitions of the associated structures

In this section we also partition the associated structures of the main tree. This makes it possible to partition the structure in parts of size o(n).

For similar reasons as in Section 2.2, we have to modify the definition of range trees.

Let $S=\{p_1 \le p_2 \le p_3 \le ... \le p_n\}$ be a subset of the 2-dimensional real vector space, ordered according to their x-coordinates.
Partition S into sets $S_1=\{p_1,p_2,...,p_{h(n)}\}$, $S_2=\{p_{h(n)+1},...,p_{2h(n)}\}$,..., where $h(n)=\lceil \sqrt{n} \rceil$.

Definition 6 : A balanced range tree, representing the set S, is defined as follows.

(i) Each set S_i is represented by an ordinary 2-dimensional range tree T_i.
Let r_i be the root of T_i. These roots are ordered according to $r_1<r_2<r_3<...$

(ii) The roots r_i are stored in a perfectly balanced augmented BB[α] leaf search tree T.
Let v be an internal node of T, representing the roots $r_i,r_{i+1},...,r_j$.
Then v contains an associated structure representing the set $S_{ij}=S_i \cup S_{i+1} \cup ... \cup S_j$, which has the following form. Let m be the cardinality of S_{ij} ($m=\Omega(\sqrt{n})$).
We order $S_{ij}=\{q_1 \le q_2 \le ... \le q_m\}$ according to their y-coordinates, and we partition it into sets $S_{ij1}=\{q_1,...,q_{h(n)}\}$, $S_{ij2}=\{q_{h(n)+1},...,q_{2h(n)}\}$,... (again $h(n)=\lceil \sqrt{n} \rceil$).
Then we store each set S_{ijk} in a BB[α] leaf search tree T_{ijk} (ordered of course according to their y-coordinates).
The roots of the trees T_{ijk} are stored in a perfectly balanced leaf search tree T_{ij}.
Now the trees T_{ij} and T_{ijk}, k=1,2,..., together form the associated structure of node v.

The update algorithm of this tree is similar to that of the modified range tree (see Section 2.2). Clearly, an update will take $O(\log^2 n)$ time, as long as the sizes of the sets S_i and S_{ijk} remain $\theta(\sqrt{n})$.
In order to keep the data structure balanced, we rebuild it as soon as a set S_i or S_{ijk} contains either $1/2\lceil \sqrt{n} \rceil$ or $2\lceil \sqrt{n} \rceil$ points. Note that this means that in the worst case we have to rebuild the entire structure after about $\sqrt{n}$ updates.
So the average update time of the balanced range tree is $O(\sqrt{n} \cdot \log n)$.

Theorem 10 : A balanced range tree, representing a set of n points, can be built in O(n log n) time and requires O(n log n) space to store.
Using this tree, range queries can be solved in time $O(\log^2 n + t)$, where t is the number of reported answers.
Insertions and deletions in this tree can be performed in time $O(\sqrt{n} \cdot \log n)$, on the average.

Theorem 11 : For a balanced range tree, there exists a $(\sqrt{n},\log n)$-partition.

Proof : Consider a tree T_i. Each level of this tree, together with its associated structures, forms a part. So each tree T_i is partitioned into O(log n) parts, each of size $O(\sqrt{n})$. This gives us for all trees T_i together $O(\sqrt{n} \cdot \log n)$ parts.
The tree T, without its associated structures, forms a part (of size $O(\sqrt{n})$).
So we are left with the associated structures of the tree T.
Each such structure is partitioned into trees T_{ijk}, of size $O(\sqrt{n})$, and the tree T_{ij}, also of size $O(\sqrt{n})$ (in general this tree will be much smaller).
Since tree T contains $O(\sqrt{n})$ leaves, and since it has depth O(log n), the associated structures together are partitioned into $O(\sqrt{n} \cdot \log n)$ parts, each of size $O(\sqrt{n})$.
To summarize, we have partitioned the balanced range tree into $O(\sqrt{n} \cdot \log n)$ parts, each of size $O(\sqrt{n})$.
If we do not have to rebuild the structure, an update will change exactly one tree T_i (which is partitioned into O(log n) levels), and on each level of tree T exactly one tree T_{ij} and one tree T_{ijk}. Hence in this case an update changes O(log n) parts of the partition.
If we do have to rebuild the tree, which has to be done at most once every $\Omega(\sqrt{n})$ updates, then $O(\sqrt{n} \cdot \log n)$ parts will change. Hence, on the average, O(log n) parts will change after an update.

❑

Remark : If we use the above partition, then after an update we have to transport an amount of $O(\sqrt{n} \cdot \log n)$ data, requiring O(log n) seeks. Hence, the total time required to update the structure in secondary memory dominates the update time of the structure in core.

3. Multi-dimensional range trees

In this chapter we generalize the results of Sections 2.1 and 2.2 to multi-dimensional range trees. The first theorem generalizes Theorems 3,4 and 5 (the proof is left to the reader).

Theorem 12 : For a d-dimensional range tree, there exists

(i) an $(n \log^{d-1} n, 1)$-partition;
(ii) a $(1, \log^{d} n)$-partition;
(iii) an $(n \log^{d-2} n, \log n)$-partition.

Also the two theorems on modified range trees can easily be generalized.
The definition of a modified d-dimensional range tree is similar to the 2-dimensional case.
Again we split the set of points into subsets S_i of size $\theta(n/\log n)$, as in Section 2.2. Each set S_i is represented by an ordinary (d-1)-dimensional range tree. The roots of these trees are stored in a

perfectly balanced binary search tree T. Each internal node v of T contains an ordinary (d-1)-dimensional range tree for the points represented by v, taking only the last d-1 coordinates into account.
This modified range tree has asymptotically the same complexity as the ordinary range tree (insertions and deletions are performed in the same way as in Section 2.2).
Clearly, Theorem 7 generalizes to the next theorem (the proof is similar).

Theorem 13 : For a modified d-dimensional range tree, there exists an
$(n \log^{d-2} n, \log \log n)$-partition.

Remark : The remark following Theorem 7 also applies here. In fact, it follows in the same way as in Theorem 8, that for a modified d-dimensional range tree, there exists an
$(n \cdot \log \log n \cdot \log^{d-2} n, 2)$-partition.

Also, Theorem 9 can be generalized to the multi-dimensional case (again the proof is a straightforward generalization of the two-dimensional case, and is left to the reader).

Theorem 14 : For a modified d-dimensional range tree, there exists an
$(n \log^{d-2} n, \log^* n)$-partition.

Also the results of Section 2.3 can be generalized to multi-dimensional range trees, but the details are more complicated. See the full version [5] of this paper for more details.

4. Storage considerations

Up to now we did not consider the amount of storage used in secondary memory. It might seem that this is exactly the same amount as in main memory, because we only keep a copy of the data structure in secondary memory, but this is not true. When we change a part, we have to write it to secondary memory. This only fits in the old space for the part when the new size is not larger. But sizes of parts grow when n grows. When the part does not fit into the same slot in secondary memory, we either have to find a new slot for it or we have to split. The first solution creates holes in the structure and, hence, increases the amount of storage in secondary memory. The second solution increases the number of seeks necessary to write the part, something we clearly want to avoid. A second problem with having holes in the data structure is that, when reconstructing the structure after a crash, we either have to read the holes or jump over them, using extra seeks.

To solve this problem we will reserve larger slots for storing parts than is actually necessary. In this way, the slot will have enough room to store the part, even when it grows. To be more precise, consider an (f(n),g(n))-partition, and let F(n) be the largest size a part can have when the structure

contains n points. Clearly $F(n)=O(f(n))$. We assume that F(n) behaves smoothly in the sense that $F(O(n))=O(F(n))$, and that F(n) is non-decreasing. Now assume at some moment, at which the set contains n_0 points, we rebuild the whole data structure in secondary memory. Rather than using slots of size $F(n_0)$, we use slots of size $F(2n_0)$. As a result, as long as n (the current number of points) is at most $2n_0$, parts still fit into their slots. At the moment when $n=2n_0$, we reconstruct the entire data structure again in secondary memory.
When n becomes very small, because of a large number of deletions, the amount of storage in secondary memory also becomes too large. To avoid this, we also rebuild the entire structure when $n \leq n_0/2$.

Theorem 15 : The shadow administrations can be stored using O(S(n)) storage, without increasing the average update costs in order of magnitude.

Proof : The number of parts is $O(S(n)/f(n))$. Each part requires $F(2n_0) \leq F(4n)=O(F(n))=O(f(n))$ storage. The storage bound follows.
When the entire structure has to be rebuilt, there must have been $\Omega(n_0)$ updates. Clearly, the rebuilding of a structure of n points takes time at most the time required for n insertions. As $n=O(n_0)$, the average update time will never be increased by more than a constant factor.

❑

A second problem with storage might be that the physical block size is larger than the slots we need. This will occur when n is small and/or the parts are small (like in Section 2.3). In this case we can pack a number of slots into one physical record in the usual ways for structures in secondary memory.

5. Conclusions and extensions

In this paper we have given a number of methods for partitioning range trees, such that updates change only a small number of parts. This enables us to store range trees in secondary memory and maintain them efficiently. This is very useful in the case the structure does not fit in main memory, or when we want to maintain a shadow administration. The results show that in such cases only small parts of the data structures have to be changed.

We treated the partitioning problem in an abstract frame work and did not go into implementation details. In a full version of this paper [5], we will also consider such problems as how to determine which parts have to be rewritten, etc.

This paper is just a first step in considering partitions of data structures. Other, more complicated, partition schemes for range trees do exist and, in fact, the bounds in this paper can be improved in