Optimal Dynamic Range Searching in Non-replicating Index Structures*

K. V. Ravi Kanth and Ambuj K. Singh
Computer Science Department
University of California at Santa Barbara
Santa Barbara, CA 93106

Abstract

We consider the problem of dynamic range searching in tree structures that do not replicate data. We propose a new dynamic structure, called the O-tree*, that achieves a query time complexity of* $O(n^{(d-1)/d})$ *on* n *d-dimensional points and an amortized insertion/deletion time complexity of* $O(\log n)$*. We show that this structure is optimal when data is not replicated. In addition to optimal query and insertion/deletion times, the O-tree also supports exact match queries in worst-case logarithmic time.*

1 Introduction

Given a set S of d-dimensional points, a range query q is specified by d 1-dimensional intervals $[q_i^s, q_i^e]$, one for each dimension i, and retrieves all points $p = (p_1, p_2, \ldots p_d)$ in S such that $\langle \forall i \in \{1, \ldots, d\} : q_i^s \leq p_i \leq q_i^e \rangle$. This type of searching in multidimensional space has important applications in geographic information systems, image databases, and computer graphics. Several structures such as the range trees [3], P-range trees [29] have been designed to support range searching in polylogarithmic time. These structures require non-linear storage space, which may not be acceptable for large database systems. Besides, updates in these structures have to handle multiple copies of data points. This, in turn, leads to consistency problems in concurrent environments. Consequently, most commercial database systems such as Illustra opt for non-replicating index structures for supporting multidimensional data. In this paper, we focus on non-replicating index structures for multidimensional range searching. Non-replicating index structures store each data point exactly once. These structures are easy to update and have low storage requirements. Examples of such structures are R-trees [1, 4, 17], Bang files [15], Grid files [21], Quad trees [27], and the divided k-d trees [31]. In this paper, we propose a new dynamic structure that achieves $O(n^{(d-1)/d})$ range query time complexity and

*Work supported in part by a research grant from NSF/ARPA/NASA IRI–9411330 and NSF instrumentation grant CDA-9421978.

$O(\log n)$ insertion/deletion time complexity. This structure improves the best known bound — $O(n^{(d-1)/d}\log^{1/d} n)$ of the divided k-d tree [31] for query time complexity in dynamic environments. We also show that this structure is optimal when data is not replicated.

Several results for the complexity of multidimensional range searching have been established in computational geometry literature. Fredman [12, 13, 14] analyzed the complexity of range searching in dynamic environments. Yao [32] analyzed it for semi-dynamic environments. In static environments, Vaidya [30] established a storage space-query time tradeoff that was later strengthened by Chazelle [6, 8]. These analyses estimated the cost using the number of instructions and did not consider memory accesses. In large databases, the memory/disk accesses are more expensive than computation. Considering this, Chazelle [6], Subramanian et al. [29], and Vaidya [30] analyzed the complexity in terms of the number of memory/disk accesses. Using a pointer-machine model, Chazelle [6, 7, 9] showed that polylogarithmic query time for n d-dimensional points can only be obtained with $O(n(\log n/\log\log n)^{d-1})$ space. His "filtering search" technique [5], which used $O(n\log n/\log\log n)$ space for 2-dimensional data and achieved logarithmic query time, was optimal. Subramanian et al. [29] also considered the memory/disk block size, B, and enhanced Chazelle's results for 2-dimensions. They showed that $O(\log_B^c n + t/B)$ query time, where c is any positive integer and t is the size of the query result, can only be achieved with $O(\frac{n}{B}(\log\frac{n}{B})/\log\log\frac{n}{B})$ storage space. For arbitrary number of dimensions, Vaidya [30] established a storage space (S)-query time (T) tradeoff equation for static environments:

$$S \geq \frac{c_1 n \log^{d-1} n}{c_2\kappa(d,n)T(\log T + \log\log n)^{d-1}}.$$

Here $\kappa(d,n)$ is 1 for $d=2$ and $\log n$ otherwise. Time T is the query time overhead per retrieved object. For linear-space structures, this equation predicts a lower bound of $\log^{d-1} n$ for query time. However, the only structure that comes any close to this bound is Bentley's non-overlapping k-level d-range structure [3] for d-dimensional data. This structure achieves $O(n^{(d-1)/k}\log n)$ query time. By choosing k appropriately, $O(n^\epsilon)$ query time can be achieved, for any $\epsilon > 0$. However, this structure maintains $O(k^{d-1})$ copies of each data point to achieve this complexity. Furthermore, the query time becomes $O(n)$ when no such replication is allowed. We investigate the complexity of non-replicating index structures. In this context, Mehlhorn [20] shows that query time complexity of partial match queries (a restricted form of range queries where only a few of the intervals are specified; the others are assumed to be $[-\infty, +\infty]$) is bounded below by $\Omega(n^{(d-1)/d})$ using a decision-tree model. We show that this lower bound also holds in a much general tree structure model.

Next, we focus on obtaining a non-replicating index structure that has optimal query time complexity in dynamic environments. Current structures including R-trees [1, 4, 17], Bang files [15, 16], grid files [21], and hB-trees [19] have a worst-case range query complexity of $O(n)$ for n data points. Commercial spatial engines such as ESRI's SDE [11] and Oracle's SDO [23] also have $O(n)$ worst-case query time complexity. The pseudo quad and k-d trees [26] achieve a query time of $O(n^{((d-1)/d)(1+\epsilon)})$ for any $\epsilon > 0$ and $O(\log^2 n)$ amortized time for insertions and deletions. The divided k-d tree [31] improves this query time to $O(n^{(d-1)/d}\log^{1/d} n)$ and insertion/deletion time to $O(\log n)$. We propose a new structure – the *O-tree* (short for "optimal k-d tree") that achieves $O(n^{(d-1)/d})$ range query time and

$O(\log n)$ amortized insertion and deletion times. The structure combines the static k-d tree [2] and the divided k-d tree [31] in a novel fashion and achieves optimality for both queries as well as updates. As in the divided k-d tree, the amortized times for insertions/deletions can be transformed into worst-case bounds using replication. In addition, the O-tree also supports exact match queries in worst-case logarithmic time.

Related to range searching in multidimensional index structures is the retrieval of multidimensional data items. Hellerstein et al. [18] study the effect of blocking strategies for d-dimensional data items. For any query that retrieves k data items, they obtain lower bounds on the worst-case number of blocks accessed. Nodine et al. [22] study different ways of minimizing these block accesses assuming sufficient amount of buffering. Note that both these papers address the problem of the complexity of data retrieval. They do not estimate the overheads incurred by a query before retrieving relevant data. Estimation of such overheads is the primary focus of this paper.

The rest of the paper is organized as follows: In Section 2, we obtain a lower bound for range query time complexity in non-replicating index structures. In Section 3, we examine different organizations for 2-dimensional data to minimize query and update (insertion/deletion) overheads simultaneously. This investigation leads to the design of the O-tree. In Section 4, we generalize the O-tree to arbitrary dimensions and explain how insertions and deletions are supported. We summarize these results in the concluding section.

2 Lower Bound for Range Query Time Complexity in Non-replicating Index Structures

2.1 Model

Following Vaidya [30], we model index structures as rooted trees. Each vertex v in such an index stores a set of data items (denoted by *data(v)*) and has at most a constant number, say r, of children (denoted by $children(v)$). Each edge $\langle u, v\rangle$ from a vertex u to a child v is associated with a condition, denoted by $cond(v)$. This condition is a conjunction of binary comparisons, each involving an endpoint of an arbitrary query and a constant. Let $\langle [x_1^s, x_1^e], \ldots, [x_d^s, x_d^e]\rangle$ denote an arbitrary query. Then, $cond(v)$ is expressed as a conjunction of d sub-clauses $C_1, C_2, \ldots, C_d$, where each sub-clause C_i is a conjunction of comparisons involving only the query endpoints x_i^s and x_i^e. For example, an edge in a 2-dimensional index structure may have the condition $C_1 \wedge C_2$, where $C_1 = (x_1^s < 4) \wedge (x_1^e > 2)$ and $C_2 = (x_2^s < 10) \wedge (x_2^e > 3)$. A structure is *non-replicating* if every data item is stored in *exactly* one vertex. Given this structure, a range query q is specified by d 1-dimensional intervals $[q_i^s, q_i^e]$, $1 \leq i \leq d$, and is denoted by $\langle [q_1^s, q_1^e], [q_2^s, q_2^e], \ldots, [q_d^s, q_d^e]\rangle$. Processing of the query starts at the root vertex and navigates to all *relevant* vertices in the tree. A vertex v is *relevant* for query q if q satisfies the condition $cond(v)$. In that case, the query visits vertex v and searches in the associated data items, *data(v)*, at v. The query then propagates to all relevant children of v. Let $U(q)$ denote the set of vertices thus visited, i.e., $U(q)$ is the set of relevant vertices whose ancestors are also relevant for query q. Then, the query time complexity is proportional to the *query access cost*, which is measured as $|U(q)| + \Sigma_{v \in U(q)} |data(v)|$.

The above model differs from Vaidya's model in that it does not allow disjuncts of comparisons in the edge conditions. In contrast to Mehlhorn's decision tree model which has d decision trees, one for each dimension, our model has one structure. We note that our model represents most current linear-space index structures including R-trees [1, 4, 17], Bang files [15], Quad-trees [27] and is much more general than Mehlhorn's decision trees [20]. In what follows, we obtain a lower bound for range query time complexity in this model.

2.2 Lower Bound

Since a lower bound in a restricted domain also holds for much general ones, we restrict the data points to have only integer values for their coordinates. Without loss of generality, the constants in edge conditions are also assumed to be integer values. Let T be a non-replicating index structure for such data. Let $D(v)$ denote the set of data points in the subtree rooted at a vertex v in T. Let the cardinality of this set be denoted by $n_D(v)$. Define a point-query at $l = (l_1, \ldots, l_d)$ to be a restricted range query $\Psi_d(l) = \langle [l_1, l_1], \ldots [l_d, l_d] \rangle$. The subscript d indicates the dimensionality of the argument l.

To obtain a lower bound for range query time complexity in non-replicating index structures, we work with the following layout of n d-dimensional points. For simplicity, let $n = k^d$.

$$S_d^n = \{(2a_1 - 1, 2a_2 - 1, \ldots, 2a_i - 1, \ldots, 2a_d - 1) \mid a_i \in \{1, \ldots, k\}, \forall i \in \{1, \ldots, d\}\}$$

This set S_d^n forms a d-dimensional grid of length $k = n^{1/d}$ in each dimension with a 2-unit separation between every two adjacent vertices. We omit the superscript n, wherever possible, for clarity. For any subset S of S_d, we define an i-projection set $P_i(S)$ as the set of X_i-coordinates of all points in S, i.e., $P_i(S) = \{p_i \mid (p_1, \ldots, p_d) \in S\}$. Such a projection set has a *gap* at x if:

- x lies in the range of $P_i(S)$ (i.e., $\exists a, b \in P_i(S) : a < x < b$), and,
- x is *even*.

We concentrate on the following set of queries $Q = \bigcup_{1 \le i \le d} Q_i$, where Q_i is the set of queries with the following specification:

- the interval along the i^{th} dimension X_i is : $[2c, 2c]$ for some c: $1 \le c \le k$, and,
- intervals in all other dimensions are: $[1, 2k + 1]$.

Note that none of the queries in Q retrieve any data from the set S_d. Hence, they are referred to as *hole* queries in general, and as i-hole queries if they are specifically from Q_i. Also note that the projection of an i-hole query to the dimension X_i corresponds to a gap. Using these observations, we show that there exists a hole query in any non-replicating index structure for S_d that has a query complexity of $O(n^{(d-1)/d})$.

Lemma 1 *Let T be a non-replicating index for the set of points S_d^n. Then, the total access cost for all the hole queries in T is at least $n - o(n)$.*

Proof: Details are given in Appendix B. □

Theorem 1 *The time complexity of range queries in a non-replicating index structure for n d-dimensional points is $\Omega(n^{(d-1)/d})$.*

Proof: Let $n = k^d$. Let T be any non-replicating index structure for the set of points S_d^n. Then, from Lemma 1 the total query cost in T for all the holes is at least $n - o(n)$. Since the number of hole queries is at most $d(n^{1/d} - 1)$, by pigeonhole principle it follows that there is a hole query that has time complexity of $O(n^{(d-1)/d})$. □

Bentley's k-d trees [2] (also dd-trees of [20]), achieve this query time complexity in static databases.

Theorem 2 *There exists a non-replicating index structure for n d-dimensional points that has $O(n^{(d-1)/d} + k)$ query time for queries retrieving k data points. Such a structure can be constructed in $O(n \log n)$ time for n points.*

3 Optimal Non-replicating Index Structure for Range Searching in Dynamic Environments

In this section, we first examine current dynamic structures for range searching and then propose a new structure with optimal query and insertion/deletion time complexity. Popular linear-space structures such as R-trees [17, 1, 4], Bang files [15, 16] support insertions/deletions in logarithmic time. However, the query time complexity in these structures is linear in the number of points.

Theorem 3 *The worst-case range query time complexity of an R-tree for n d-dimensional points is $O(n)$.*

Proof: Details are given in Appendix A. Proofs for other structures are also included there. □

The R-trees and the k-d trees are two extreme solutions for the problem of range searching in dynamic environments. While the R-trees support insertions/deletions in logarithmic time, the k-d trees support queries in optimal time and insertions/deletions in $O(n \log n)$ time (to preserve query optimality). Several attempts to bridge this gap have been made. The pseudo k-d trees [26] try to reduce the insertion/deletion overhead of k-d trees. They achieve an amortized insertion/deletion complexity of $O(\log^2 n)$. The divided k-d trees [31] reduce the insertion/deletion overheads further down to $O(\log n)$. In their attempt to support insertions/deletions efficiently, these and other dynamic structures compromise query optimality. For instance, the query time complexity becomes $O(n^{((d-1)/d)(1+\epsilon)})$ in pseudo k-d trees and $O(n^{(d-1)/d} \log^{1/d} n)$ in the divided k-d trees. As a result, there exists no dynamic structure that combines optimal range query time with logarithmic insertion/deletion times. We now propose a new structure, called the *O-tree*, that accomplishes this task. Besides answering range queries in optimal time, the O-tree supports insertions and deletions in $O(\log n)$ amortized time. In what follows, we describe the design principles of the O-tree for 2-dimensions. We first introduce some preliminary concepts for organizing 2-dimensional data in Section 3.1. Then, we describe the C-tree, a dynamic structure to support range queries in optimal time and insertions and deletions in $O(\log^2 n \log \log^2 n)$ time, in Section 3.2. In Section 3.3, we refine the C-trees to O-trees to reduce insertion/deletion times to $O(\log n)$. Note that for non-replicating tree structures, this insertion/deletion complexity is optimal.

3.1 Preliminaries

3.1.1 Definitions

A 1-dimensional interval I is specified by a starting point (denoted I^s) and an ending point (denoted I^e) in a linear domain. Interval I is *contained* in interval J if $((I^s \geq J^s) \wedge (I^e \leq J^e))$. Interval I *intersects* interval J if $((I^s \leq J^e) \wedge (J^s \leq I^e))$. Note that containment is a special case of intersection. Next, we extend these definitions to 2-dimensions. A 2-dimensional interval object I is specified by 2 1-dimensional intervals, one in the x-dimension and the other in the y-dimension. This object I is *contained* in another 2-dimensional object J if the containment holds in each dimension. Similarly, it *intersects* J if the intersection holds in each dimension. We say a 2-dimensional interval object I is contained in another object J in a specific dimension if the interval of I in that dimension is contained in the corresponding interval of J. These definitions can be easily generalized to arbitrary dimensions.

3.1.2 Partitioning 2-dimensional Data

Consider a set S of 2-dimensional points. These points can be ordered using their x-coordinates. If the x-coordinates match, the points can be ordered using their y-coordinates. Alternately, they can be ordered using their y-coordinates first, and then using their x-coordinates. Let the former ordering be referred to as the x-order and the latter the y-order. We denote that point p_1 precedes point p_2 in x-order by $p_1 \prec_x p_2$. Likewise, $p_1 \prec_y p_2$ denotes the precedence of point p_1 over p_2 in y-order. Let S_σ denote the sequence of points in S sorted using σ-order, where σ denotes either the x or the y-dimension. The extent of such a sequence can be represented by the first and last points in the sequence. This extent denotes the extent of the set S in σ-order. Alternately, the extent of the set S can be represented by a *bounding rectangle*, denoted by $b(S)$. This bounding rectangle has an interval for each dimension. The interval in x-dimension is denoted by $b_x(S)$ and is obtained by taking the minimum and maximum values of the x-coordinates of the points in the set S. Likewise, the interval in y-dimension is denoted as $b_y(S)$ and is obtained using the y-coordinates. To illustrate these concepts, consider an example. Let $S' = \{(1,1),(2,6),(2,4),(3,5),(5,1),(6,6),(6,4),(7,5)\}$. The x-order orders the points in S using the x-coordinates. Points that have same x-coordinates, for instance $(2,4)$ and $(2,6)$ (or $(6,4)$ and $(6,6)$), are ordered using the y-coordinates. The extent of the set S' is the first and last points in this sequence, which are $(1,1)$ and $(7,5)$ respectively. The minimum and maximum x-coordinates for points in S' are 1 and 7. Therefore, the x-interval of the bounding rectangle for S' is $b_x(S') = [1,7]$. Similarly, the minimum and maximum y-coordinates are 1 and 6. Therefore, the y-interval is $b_y(S) = [1,6]$.

Total orders, such as $\prec_x$ and $\prec_y$ above, can be used to partition a set S into smaller subsets that are disjoint and are of approximately same size. We refer to such subsets as partitions of S in the specified order. For example, the set S' can be divided into 4 partitions, $P_1,\ldots,P_4$ using x-order as:

$$P_1 = \{(1,1),(2,4)\}, P_2 = \{(2,6),(3,5)\};$$

$$P_3 = \{(5,1),(6,4)\}, P_4 = \{(6,6),(7,5)\}.$$

A total order on individual points can be extended to the extents of partitions of a set S. Partition extent $\langle a, b \rangle$ precedes another partition extent $\langle c, d \rangle$ in σ-order if b precedes c in σ-order.

3.1.3 B-tree for Partitions of 2-dimensional Data

Let P denote the set of partitions for a set of 2-dimensional points obtained using σ-order. Since the extents of the partitions in P can be totally ordered using σ-order, a B-tree (the variant that stores data in the leaves) can be constructed over the partitions using these extents. We refer to this tree as a B-tree for the partitions in σ-order, and denote it by $B(|P|, \langle\sigma\rangle)$. Figure 1(a) shows such a tree for the partitions of S'. Such a tree facilitates efficient location of necessary partitions. For instance, identifying a partition in which a new point s is to be stored can be accomplished as follows. We start at the root of the B-tree and traverse down the tree by choosing the i^{th} entry (corresponding subtree) such that $p_1^i \prec_\sigma s \prec_\sigma p_1^{i+1}$, where $\langle p_1^i, p_2^i \rangle$ and $\langle p_1^{i+1}, p_2^{i+1} \rangle$ are the extents associated with the i^{th} and $(i+1)^{st}$ entries respectively. For example, for inserting a new point $(4, 1)$, we choose the right subtree at the root and then the partition P_2 at vertex v. In general, for a tree of k partitions, this procedure takes $\log k$ time.

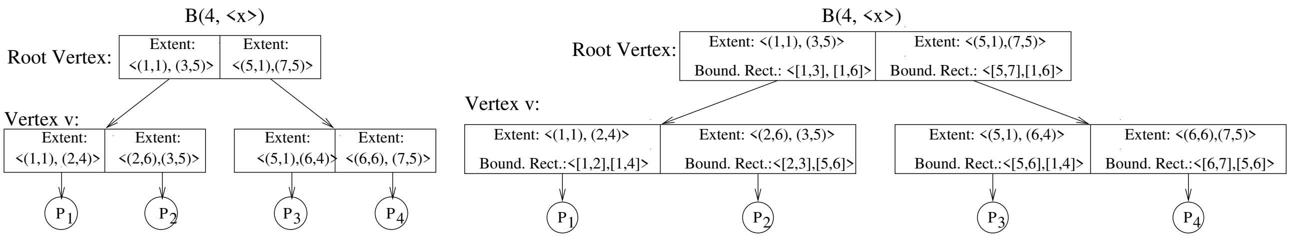

Figure 1. (a)B-tree for 4 partitions of S' using x-order (b) Augmented with bounding rectangles

To ensure that the B-tree fits the model of Section 2 for processing range queries, we augment it as follows. Each index entry is associated with a bounding rectangle for all the points in the corresponding subtree. The edge condition associated with the index entry is then specified as an intersection of an arbitrary query with the bounding rectangle. For instance, if the bounding rectangle has interval $[l_i, m_i]$ in dimension i, the conjunction C_i of the edge condition is expressed as $(x_i^s \leq m_i) \wedge (x_i^e \geq l_i)$. Figure 1(b) shows the augmented B-tree using x-order for the set S'. The bounding rectangle for the set $P = P_1 \cup P_2$ is $\langle [1, 3], [1, 6] \rangle$. This is stored in the first entry of the root vertex leading to the subtree containing P_1 and P_2. The bounding rectangles for the other entries can be similarly explained. Hereafter, we assume every B-tree for a set of partitions is augmented with bounding rectangles.

Given this structure, a query $q = \langle q_x, q_y \rangle$ accesses vertex v from its parent u only if it intersects the bounding rectangle associated with the edge $\langle u, v \rangle$. For example, in Figure 1(b) a query accesses vertex v from the root vertex only if it intersects the bounding rectangle $\langle [1, 3], [1, 6] \rangle$ associated with the entry for v in the root vertex. Since checking for intersection involves a conjunction of binary comparisons, this structure satisfies the model of Section 2. Before proceeding further, we prove the following property.

Lemma 2 *In a B-tree for a set of partitions using σ-order, at most 2 partitions that intersect a query are not contained in the query in dimension σ.*

Proof: In Appendix C. □

3.2 C-tree: An Optimal Structure for Queries

Let us now examine how best to organize an arbitrary 2-dimensional data set S. Assume it is partitioned into k subsets using y-order. The best value of k for a set S of n points will be determined during the course of the discussion. Figure 2(a) shows the partitions $P_1, \ldots, P_k$ for the 2-dimensional set S. By maintaining a B-tree for these partitions, we can locate partitions intersecting a query $q = \langle q_x, q_y \rangle$ efficiently. Of these intersecting partitions, there are at most two *extreme* partitions which are not contained in q in the y-dimension. Partitions P_i and P_j are such extreme partitions in Figure 2(a) for the query q. All other intersecting partitions are contained in the query interval q_y and hence are referred to as *interior* partitions. Partitions P_{i+1} to P_{j-1} are interior partitions. In these partitions, we have to search just using the query interval q_x. This search can be accomplished efficiently if the data in these partitions is ordered according to their x-coordinates. Hence, each partition is subdivided into m sub-partitions using x-order. The sub-partitions for each partition are organized as a B-tree. Figure 2(b) shows the resulting organization. We have two layers of B-trees. The top layer (referred to as layer-2 tree) has k partitions using y-order. Each partition of this tree is organized as a layer-1 B-tree of m partitions using x-order.

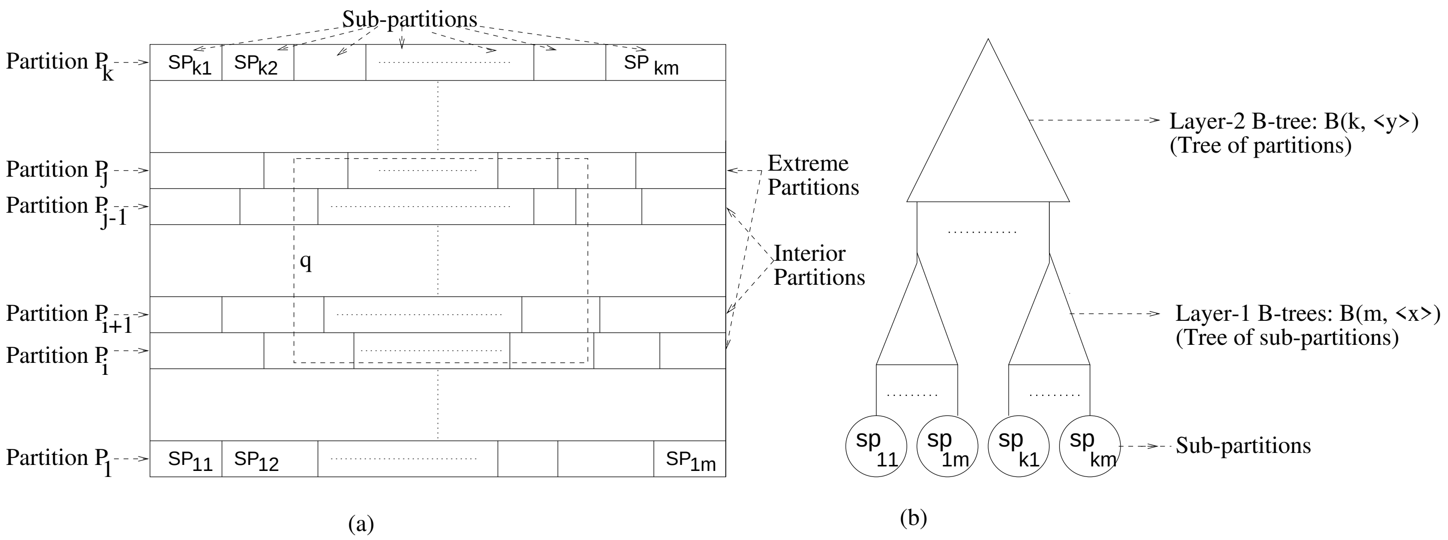

Figure 2. (a) Partitions, sub-partitions for 2-dimensional data (b) 2-layered tree over the sub-partitions

3.2.1 Range Queries

Let us examine how range queries are processed in the above organization. As specified in the model of Section 2, a query $q = \langle q_x, q_y \rangle$ starts at the root of the layer-2 B-tree and accesses a vertex v only if the corresponding edge condition is satisfied, i.e., only if q intersects the associated bounding rectangle. To analyze the worst-case time complexity for the query, we trace the execution of the query as follows:

1. Find all the partitions intersecting query q in the layer-2 B-tree.
2. For each interior partition (the y-interval is contained in q_y), such as $P_{i+1}, \ldots, P_{j-1}$ in Figure 2, search in the layer-1 B-tree as follows:
   (a) Find all sub-partitions that intersect q in the corresponding B-tree.
   (b) For the interior sub-partitions, i.e., for the sub-partitions with x-intervals contained in q_x, retrieve all associated data. All such data form part of the query result.
   (c) For the extreme sub-partitions, search in the sub-partitions.
3. For the extreme partitions, such as P_i and P_j in Figure 2,
   (a) Find all sub-partitions that intersect q in the corresponding B-tree.
   (b) Search in these sub-partitions.

Let us now analyze the worst-case overhead, t, for a query q. Step 1 of the algorithm takes $O(k)$ time, since the B-tree has at most k partitions. This is because some partitions may intersect the query in the y-dimension and not in the x-dimension. Let $n_p \leq k$ partitions intersect the query in both dimensions. Of these at most two of them can be extreme partitions, i.e., the y-intervals of their bounding rectangles are not contained in the corresponding query interval. Consequently, step 2 is executed $(n_p - 2)$ times, whereas step 3 is executed at most 2 times. For each of the $(n_p - 2)$ interior partitions, step 2(a) has overheads of $\log m$ in the corresponding layer-1 B-tree. Note that in this case, logarithmic overheads are achieved as opposed to linear overheads in Step 1. This is because of the following reasons. First, all sub-partitions intersecting q are stored as a linear sequence using the x-order. Second, all these sub-partitions, being parts of an interior partition, are contained in the query in the y-dimension (i.e., q_y contains their corresponding y-intervals). Next, note that the interior sub-partitions are contained in the query in both x- and y-dimensions. Hence, all data points in such sub-partitions are part of the query result. Assuming they are organized using space linear in the number of points, the retrieval cost in these sub-partitions is directly proportional to the query result. Hence, they do not contribute to the query overheads.

For the extreme sub-partitions in step 2(c), we continue the search in the sub-partition using the query q. Since each of these sub-partitions has $O(n/(km))$ data points, let us denote the associated overheads by $ST(n/(km))$. This cost is also borne out for the sub-partitions of the extreme partitions (step 3(b)). Note that the cost of finding these sub-partitions in each extreme partition is $O(m)$. Summing up, the total overhead t is given by:

$$t = k+(k-2)*[\log m+2*ST(n/(km))]+2*[m+m*ST(n/(km))] = k+(k-2)\log m+2m+[2(k-2)+2m]*ST(n/(km))$$

By organizing the sub-partitions in different ways and finding the optimal values for k and m, we achieve different structures. We consider three such organizations for the sub-partitions. The first two, will be discussed in this subsection. The third yields the O-tree and is described in the Section 3.3.

Assuming that the search in the sub-partitions involves a linear scan of all the data points (i.e., $ST(l) = l$), the above equation yields a minimum value of $O(\sqrt(n \log n))$ for t, when $k = m = O(\sqrt(n/\log n))$. The same overhead is also obtained when $k = O(\sqrt(n/\log n))$ and $m = O(\sqrt(n \log n))$. This latter organization corresponds to the divided k-d tree of Kreveld and Overmars. As a side-note, note that when k and m are constants and the 2-layered tree organization is repeated $O(\log n)$ times, we obtain the static k-d tree of Bentley [2].

Consider an alternative organization for the data in sub-partitions of Figure 2 to reduce the search costs. Assume each sub-partition is organized as a static k-d tree. Then the search cost in a sub-partition of size $c = O(n/(km))$ reduces from c in the divided k-d tree to $\sqrt{c}$. Consequently, the query overhead becomes:

$$t = k + (k-2)\log m + 2m + [2(k-2) + 2m] * O(\sqrt(n/(km)))$$

This equation has minimum value of $O(\sqrt{n})$ when $k = m = O(\sqrt{n}/\log n)$. Note that this query overhead corresponds to the lower bound (for 2-dimensional data) obtained in Section 2. Hence, this structure is optimal for range queries. We refer to this structure as the combined k-d tree, C-tree for short, since it combines the divided k-d tree with the static k-d trees. As in the divided k-d tree [31], the C-tree has 2-layers of B-trees, each organized on a different dimension. However, at both the layers, every B-tree has $k = O(\sqrt{n}/\log n)$ partitions. The exact value of k will be specified later. We refer to these two layers as a 2-layered B-tree of k partitions. In addition, the C-tree has a third layer that is organized as static k-d trees. We refer to the trees in the third layer as the *leaf partitions of the 2-layered B-tree*. Figure 3 shows the organization of the C-tree. It has a 2-layered B-tree of $O(\sqrt{n}/\log n)$ partitions. The leaf partitions of this 2-layered B-tree have $O(\log^2 n)$ points each and are organized as 2-dimensional static k-d trees. This organization can be denoted as:

C-tree for n 2-dimensional points = 2-layered B-tree of $O(\sqrt{n}/\log n)$ partitions + 2-dimensional static k-d tree of $O(\log^2 n)$ points.

3.2.2 Exact Match Queries

Next, let us analyze the complexity of exact match queries in a C-tree. This is useful for describing and analyzing insertions and deletions later. Given a specific location p, an exact match query retrieves all data points in the C-tree located at p. This can be accomplished by first identifying a sub-partition in which p may be stored and then searching in that sub-partition, which is organized as a static k-d tree. Since each sub-partition has $O(n/(km)) = O(\log^2 n)$ points, the latter part of the search can be accomplished using the corresponding algorithm for a static k-d tree in $O(\log \log^2 n)$ time. Next, we describe the time for locating a sub-partition in which p could be stored.

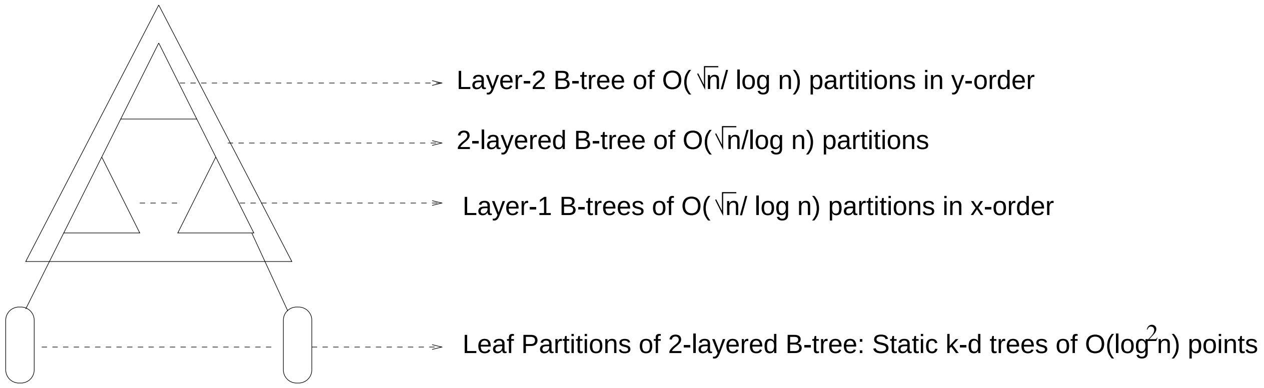

Figure 3. The C-tree for n 2-dimensional points

1. Start with the root v of the layer-2 B-tree.

2. Search in the tree rooted at v using p as follows:

   (a) Let v have m entries $\langle [e_1, c_1], \ldots, [e_m, c_m] \rangle$ ordered using y-order. Then, choose the i^{th} entry $([e_i, c_i])$ such that the extent e_i precedes p and p precedes e_{i+1} in y-order. If no such entry exists, then choose the first entry.

   (b) Repeat the above step for the corresponding child c till we reach a partition P of the B-tree.

3. The partition P is organized as a layer-1 B-tree. Repeat Step 2 using this tree to identify the sub-partition in which the insertion/deletion has to occur. Note that since the layer-1 B-tree uses x-order, the search criterion in Step 2(a) also uses x-order.

Note that unlike range queries, the above algorithm for sub-partition identification uses the extents in the B-tree. Since the disjointness of the partitions is reflected in their extents, only one entry is chosen in Step 2(a). These extents are stored in sorted order and each B-tree has at most $O(\sqrt{n}/\log n)$ partitions. Therefore, Step 2 costs $O(\log n)$ time. The same process is also repeated in Step 3. The total time complexity of this algorithm is $O(\log n)$. Hence, sub-partition identification and subsequently exact match queries are answered in the C-tree in logarithmic time.

3.2.3 Insertions and Deletions

Let us now analyze the impact of organizing each sub-partition as a static k-d tree on the complexity of insertions and deletions. Insertions and deletions are performed as follows:

1. Identify the sub-partition in which the insertion/deletion has to occur.

2. Perform the insertion/deletion in the static k-d tree corresponding to the identified sub-partition.

3. Re-balance the trees as necessary.

From Section 3.2.2, we know that the first step takes $O(\log n)$ time. Next, we analyze the complexity of Step 2. As mentioned earlier, the static k-d tree can support insertions and deletions without degrading query performance only if it is reconstructed after each insertion/deletion. Consequently, insertion in each sub-partition takes $O(c \log c)$ time, where c is the number of points in the static k-d tree. Since this number is $O(n/(km)) = O(\log^2 n)$, each insertion/deletion takes at least $O((\log^2 n)\log(\log^2 n))$ time in Step 2. In Step 3, re-balancing of the trees can be accomplished using the technique of *partial rebuilding* [24]. As explained later, this technique costs $O(\log n)$ amortized time, which can be transformed into worst-case time using the technique of *global rebuilding* [25]. Thus, a C-tree for n points supports range queries in optimal time and insertions/deletions in $O(\log n + (\log^2 n)\log(\log^2 n))$ time. This is an improvement over current index structures with optimal range query complexity. In the next subsection, we propose the O-tree that achieves optimal query performance as well as logarithmic insertion and deletion times. As in the C-tree, the insertion/deletion times are amortized bounds for O-trees.

3.3 The O-tree: An Optimal Structure for Queries, Insertions and Deletions

Consider the C-tree for n points. Range queries are answered in optimal time in this structure. The insertion/deletion time, however, is non-optimal and has two principal components – $O(\log n)$ complexity in the 2-layered B-tree and $O((\log^2 n)\log(\log^2 n))$ complexity in the static k-d trees. If we used only a static k-d tree, it would have an optimal query complexity and $O(n \log n)$ insertion/deletion time. Note that by going from a static k-d tree to a C-tree, we have successfully reduced the insertion/deletion time from $O(n \log n)$ to $O((\log^2 n)\log(\log^2 n))$ without affecting the query time. Hence, a C-tree can modularly replace a static k-d tree to reduce the insertion/deletion time. Based on this observation, we organize the leaf partitions of a C-tree not as static k-d trees but as C-trees. This reduces the insertion/deletion time from $O(\log n + (\log^2 n)\log(\log^2 n))$ to $O(\log n + (\log^2 \log^2 n)\log(\log^2 \log^2 n))$. This yields a total time of $O(\log n)$ for insertions and deletions. Since this organization has optimal query and insertion/deletion times, we call it the O-tree. To summarize, the O-tree has the following structure.

O-tree(2,n) = 2-layered B-tree of $O(\sqrt{n}/\log n)$ partitions + C-tree of $O(\log^2 n)$ points.

Since a C-tree is a 2-layered B-tree followed by static k-d trees, we have

O-tree(2,n) = 2-layered B-tree of $O(\sqrt{n}/\log n)$ partitions
+ 2-layered B-tree of $O(\log n/\log\log n)$ partitions
+ 2-dimensional static k-d tree of $O(\log^2 \log^2 n)$ points.

This organization for the O-tree is shown in Figure 4. At the first level, we have a 2-layered B-tree of $O(\sqrt{n}/\log n)$ partitions. The leaf partitions of this 2-layered B-tree have $O(\log^2 n)$ points and are organized as second-level structures. Each second-level structure is a 2-layered B-tree of $O(\log n/\log\log n)$ partitions. The leaf partitions of these second-level 2-layered B-trees are organized as static k-d trees of $O(\log^2 \log^2 n)$ points at the third level.

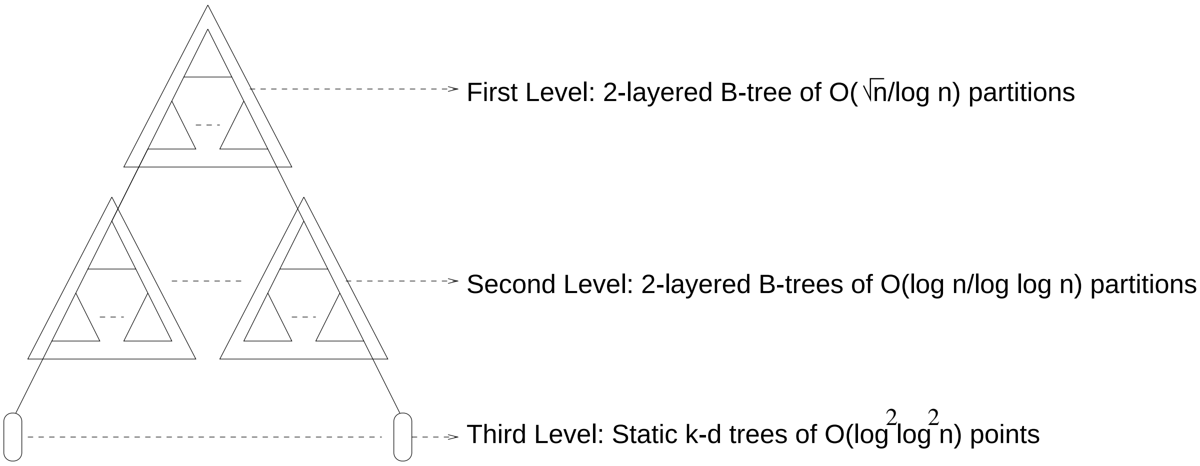

Figure 4. An O-tree for n 2-dimensional points

4 The O-tree for Arbitrary Dimensions

The 2-dimensional O-tree can be generalized to d-dimensions by making the following three changes:

1. Replace the first-level 2-layered B-tree by a d-layered B-tree of $4n^{1/d}/\log n$ partitions.
2. Replace the second-level 2-layered B-trees by d-layered B-trees of $4\log n/\log\log n$ partitions.
3. At the third level, replace each 2-dimensional static k-d tree by a d-dimensional static k-d tree of $2^{d+1}\log^d\log^d n$ points.

Note that the above replacements also have specific values for the number of partitions instead of $O()$ terms. For instance, any B-tree in the first-level d-layered B-tree has at most $4n^{1/d}/\log n$ partitions. As we see in the next section, the factor 4 allows us to incorporate insertions and deletions in the tree with logarithmic overheads. In summary, an O-tree for n d-dimensional points is organized as:

O-tree(d,n) = d-layered B-tree of $4n^{1/d}/\log n$ partitions
+ d-layered B-tree of $4\log n/\log\log n$ partitions
+ d-dimensional static k-d tree of $2^{d+1}\log^d\log^d n$ points.

To complete the above specification for the d-dimensional O-tree, we do the following.

1. Extend the total order for 2-dimensions to d-dimensions,
2. Define B-trees for partitions of d-dimensional points, and,
3. Define d-layered B-trees.

Let $X_1, \ldots, X_d$ denote the d dimensions. Then, we define X_i-order for a set of d-dimensional points as the order based on the X_i-coordinates; if these are identical, then using the $X_{(i+1) \bmod d}$-coordinates, and then using the $X_{(i+2) \bmod d}$-coordinates, and so on. Using such an order, a d-dimensional set can be divided into partitions and using their extents, a B-tree can be constructed over the created partitions. A d-layered B-tree of k partitions has d-layers of such B-trees. At layer i, each B-tree has at most k partitions in X_i-order. Figure 5 shows the structure of the resulting O-tree for n d-dimensional points. Note the similarity to the 2-dimensional case. As shown in the figure, we refer to the static k-d trees as layer-0 trees, the d layers of the second-level d-layered tree as layers 1 to d and the d layers of the first-level d-layered tree as layers $d+1$ to $2d$ of the O-tree.

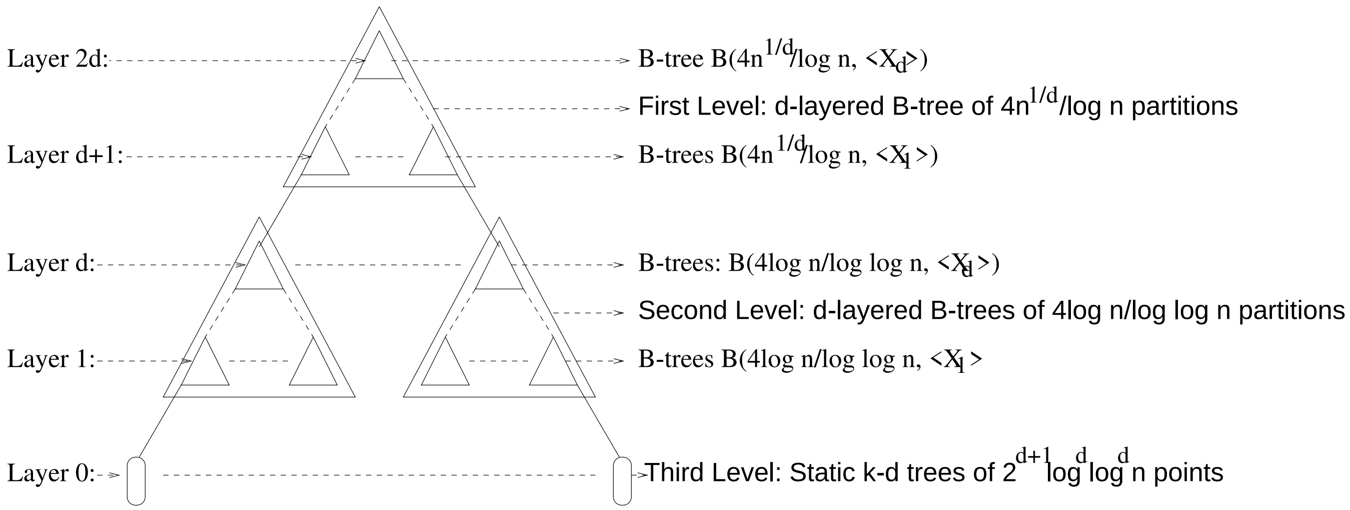

Figure 5. An O-tree for n d-dimensional points

The space complexity for an O-tree can be easily shown to be linear in the number of data points. The analysis of the range and exact match query time complexities are direct extensions of those for the 2-dimensional case.

Theorem 4 *Range queries in an O-tree of n d-dimensional points can be answered in $O(n^{(d-1)/d} + k)$ time, where k is the number of retrieved data points. Exact match queries can be answered in $O(\log n)$ time.*

Proof: In Appendix C. $\Box$

4.1 Construction of an O-tree

We now concentrate on insertions and deletions in an O-tree. We start off with the construction of an O-tree for n d-dimensional points. Then, in the following subsections we describe the insertion/deletion algorithms. For ease of description, we refer to the entire subtree rooted at a layer-i B-tree as an i-layered O-tree. The entire structure is thus a $(2d+1)$-layered O-tree. The procedure for constructing an i-layered O-tree for a set S of n d-dimensional points is given below.

Procedure *ConstructOtree(S, i, d)*

begin

1. If $(i = 0)$, construct a static k-d tree.

2. Otherwise,

   (a) If $(i > d)$, then set $k = n^{1/d} \log n$ and $j = k - d$. Otherwise, set $k = \log n / \log \log^d n$ and $k = d$.

   (b) Partition S into k subsets using X_j-order.

   (c) For each partition P of S, *ConstructOtree(P, i-1, d).*

   (d) Construct a B-tree over the k partitions of S. The root of this tree corresponds to the root of the i-layered O-tree.

end

The layer-0 trees correspond to static k-d trees, which are constructed using the algorithms of [2]. For every other layer i, the set S is partitioned into k subsets using X_j-order where k and j are determined by the value of i. Since the layers $d + 1 \leq i \leq 2d$ correspond to the layers of the first-level d-layered B-tree, the number of partitions k in each ordered tree at these layers is $n^{1/d}/ \log n$ and the ordering dimension is given by $(i - d)$. For the lower layers, $k = \log n / \log(\log n)$ and $j = d$. Once the set S is partitioned, an $(i - 1)$-layered O-tree is constructed for each of the resulting partitions. A B-tree is then built on these partitions.

Next, we analyze the time complexity of the construction of an i-layered O-tree for a set of n points. Let $CT(n, i, d)$ denote this complexity. Partitioning a set S of n points along a particular dimension takes $O(n \log n)$ time and constructing an ordered tree for k partitions takes $O(k \log k)$ time. Hence,

$$CT(n, i, d) = O(n \log n) + kCT(n/k, i - 1, d) + O(k \log k).$$

Since construction of a static k-d tree of n d-dimensional points can be accomplished in $O(n \log n)$ time, $CT(n, 0, d)$ is $O(n \log n)$. Solving the above equation using this information, we obtain $CT(n, i, d) = O(n \log n)$, for all $i : 0 \leq i \leq 2d$.

Theorem 5 *Construction of an i-layered O-tree, $0 \leq i \leq 2d$, for n d-dimensional points takes $O(n \log n)$ time.*

Corollary 1 *An O-tree for n d-dimensional points can be constructed in $O(n \log n)$ time.*

4.2 Insertions and Deletions

Consider an O-tree constructed for n_0 points. The number of points in each i-layered O-tree, $0 \leq i \leq 2d$, is denoted by $s_i(n_0)$ and is given as follows.

- When $i = 0$, the i-layered tree is the static k-d tree. Therefore, $s_0(n_0) = \log^d \log^d n_0$.
- When $1 \le i \le d$, $s_i(n_0) = \log^i n_0$.
- When $d+1 \le i \le 2d$, $s_i(n_0) = n_0^{(i-d)/d} \log^{2d-i} n_0$.

Given such a newly constructed structure of n_0 points, insertions and deletions are supported by partially rebuilding every i-layered O-tree whenever its balance is potentially disturbed. By ensuring that $\alpha_i s_i(n_0)$ updates (insertions/deletions) occur before such rebuilding, the cost is amortized over the updates. This technique for supporting insertions and deletions is called partial rebuilding [24].

The algorithm for insertion/deletion of a new point p is given in terms of a general procedure for inserting/deleting a point in an i-layered tree T of $s_i(n_0)$ points.

Procedure *Insert-Delete(T, i, p)*

begin

1. Identify the partition P of the layer-i B-tree in which p is to be inserted/deleted. Let the corresponding tree for P be T'.
2. If $\alpha_{i-1} s_{i-1}(n_0)$ insertions have already occurred in T', then reorganize P as follows:
   (a) Read all the points in the tree T'.
   (b) Partition them into two sets P_1 and P_2 using X_{i-d}-order if $i > d$, and X_i-order otherwise.
   (c) For each partition P_1 and P_2, construct an $(i-1)$-layered O-tree.
   (d) Incorporate the split of partition P in the layer-i B-tree.
3. Otherwise,
   (a) If T' corresponds to a static k-d tree, i.e. $i = 1$, then reconstruct T' taking p into consideration.
   (b) Otherwise, *Insert-Delete(T', i-1, p)*.

end

Before analyzing the time complexity of the above insertion/deletion algorithm, we first show that after each insertion (or deletion) in an O-tree of n points, the resulting tree is an O-tree of $n+1$ points (or $n-1$ points in case of deletion). We found that this can be ensured if the fractions $\alpha_0, \dots, \alpha_{2d}$ satisfy the following inequalities. Hereafter, we refer to these fractions as the *update thresholds*.

$$\alpha_{2d} < 1 - \frac{1}{\sqrt{2}}; \quad 0 < \alpha_{2d} < \alpha_{2d-1} < \alpha_{2d-2} < \dots \alpha_1 < \alpha_0 = 1. \tag{1}$$

Lemma 3 *An O-tree of n points always exhibits the following characteristics.*

1. *At layer 0, the number of points in any static k-d tree is at most $2^{d+1} \log^d \log^d n$.*

2. *The number of partitions in a layer-i B-tree is at most*

   (a) $4 * n^{1/d}/\log n$ *if* $d+1 \leq i \leq 2d$,

   (b) $4 * \log n/\log\log^d n$ *if* $1 \leq i \leq d$.

Proof: Details in Appendix C. □

Theorem 6 *Amortized insertion/deletion cost in an O-tree of n data points is $O(\log n)$.*

Proof: Details in Appendix C. □

Combining theorems 4 and 7 for query and update complexities, we have the following result.

Theorem 7 *There exists a dynamic non-replicating index structure that supports range queries on n d-dimensional points in $O(n^{(d-1)/d} + k)$ time, where k is the number of retrieved data points. Insertions and deletions are supported in $O(\log n)$ amortized time.*

Note that the amortized times for updates can be transformed into worst-case times using the technique of global rebuilding [25], which maintains multiple copies of data. More details on the global rebuilding technique along with the proof for transforming the amortized bounds to worst-case bounds are provided in Appendix D.

5 Conclusions

In this paper we examined the complexity of dynamic range searching in tree structures that do not replicate data. Such structures are of interest due to their low storage costs and ease of updation. We proposed a new dynamic non-replicating index structure that achieves $O(n^{(d-1)/d})$ query time complexity and $O(\log n)$ amortized insertion/deletion time complexity. This structure improves the previous known upper bound [31] of $O(n^{(d-1)/d} \log^{1/d} n)$ for query time in dynamic structures. We also proved that this structure is optimal when data is not replicated. In addition to optimal query time and logarithmic insertion/deletion times, the structure also supports exact match queries in worst-case logarithmic time.

Appendix

A Analysis of Range Query Time Complexity of Some Non-replicating Structures

A.1 Range Query Time Complexity of R-trees

We demonstrate that for specific data points and queries, the R-tree and its variants [1, 4, 17, 28] have a worst-case complexity of $O(n)$ for range queries on n data points. The R-tree is an extension of the B$^+$-tree for multidimensional data. As in the B$^+$-tree, data is stored in the leaves. Internal nodes store bounding rectangles to represent the area covered by the subtrees. Specifically, each node in an R-tree has between m and $M-1$ entries. Each entry is of the form – (c, br), where c refers to a child node and br is the bounding rectangle for all points in the subtree rooted at child c. The R-tree is a dynamic structure and supports insertions/deletions by *local rebuilding* [10, 25]. Local rebuilding (or balancing) denotes the technique applied to search trees so that they maintain logarithmic height during a sequence of updates. Since the R-tree is based on a B$^+$-tree structure, it employs a similar local rebuilding technique as the latter. This technique affects (updates) only nodes along a single path from the root to a leaf. Consequently, the insertion/deletion time complexity is proportional to the tree height, which is $O(\log n)$. We show that this technique for supporting insertions and deletions can result in an R-tree that has $O(n)$ range query time for n 2-dimensional data points.

Theorem A.1 *In an R-tree of n 2-dimensional points, the worst-case time complexity for a range query is $O(n)$.*

Proof: We first note that the time complexity for a range query in an R-tree is proportional to the number of nodes whose corresponding bounding rectangles intersect the query. Since every query that intersects a leaf node also intersects all its ancestors in an R-tree, the time complexity for a query is lower bounded by the number of leaf nodes that intersect the query. We prove that the number of such intersecting leaf nodes can be $O(n)$ in the worst-case, where n is the number of 2-dimensional data points in the tree.

We consider two types of clusters C_1 and C_2 and repeatedly insert these clusters in an alternate fashion. In the first cluster type C_1, all data points have their y coordinates to be either 3 or 6. In the second cluster type, C_2, the y coordinates are either 1 or 8. Each of these clusters is specified by an offset o for the x-coordinates. Specifically, the i^{th} point in cluster C_1 at offset o has the following coordinates.

- $(o + \lfloor i/2 \rfloor, 3)$ if i is even
- $(o + \lfloor i/2 \rfloor, 6)$ otherwise

This point is denoted by $C_1(o, i)$. Similarly the i^{th} point in cluster C_2 at offset o is denoted by $C_2(o, i)$ and has the following coordinates.

- $(o + \lfloor i/2 \rfloor, 1)$ if i is even
- $(o + \lfloor i/2 \rfloor, 8)$ otherwise

Let $I(c)$ and $D(c)$ denote insertion and deletion of a point c. For ease of description let $I(C, o, 1, k)$ denote insertion of a sequence of points from cluster C at offset o, i.e., $I(C, o, 1, k) = \langle I(C(o, 1), C(o, 2), \ldots, C(o, k)\rangle$. Likewise, $D(C, o, 1, k)$ may be defined.

Without loss of generality, let $n = m(2k+1)$ for some k. We consider the following sequence of insertions into an initially empty R-tree.

$S = \langle I(C_1, 0, 1, m),$

$$I(C_2, Lm, 1, M-m), D(C_2, Lm, m+1, M-m), I(C_1, 2Lm, 1, M-m), D(C_1, 2Lm, m+1, M-m)$$
$$I(C_2, 3Lm, 1, M-m), D(C_2, 3Lm, m+1, M-m), I(C_1, 4Lm, 1, M-m), D(C_1, 4Lm, m+1, M-m)$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$I(C_2, (2k-1)Lm, 1, M-m), D(C_2, (2k-1)Lm, m+1, M-m), I(C_1, 2kLm, 1, M-m), D(C_1, 2kLm, m+1, M-m)\rangle$$

First, a set of m points from a C_1 cluster at offset 0 are inserted into the R-tree. Then, a set of $(M-m)$ points from a C_2 cluster at offset Lm are inserted. The insertion of the last point in this later sequence triggers a split of the only leaf node of the R-tree. The two nodes that are formed correspond to a cluster C_1 of m points followed by a cluster C_2 of $(M-m)$ points. Let N_1 be the leftmost node in this sequence (i.e., the node corresponding to the cluster C_1) and N_2 the other node. This conformance of the leaf nodes to the two cluster types is a consequence of the area minimization split criteria of the R-tree [17]. Since the clusters themselves do not overlap (nor do their parents), other variants such as R*-trees [1] and X-trees [4] also yield the same configuration. Next, $(M-2m)$ points are deleted from the leaf node N_2. This is followed by an insertion of the next $(M-m)$ points (corresponding to a C_1 cluster at offset Lm). All these insertions occur in the leaf node N_2, which is then split into two nodes N_2' and N_3. These nodes correspond to a C_2 and a C_1 cluster respectively. Using induction, it can be shown that the above sequence of insertions and deletions yields an R-tree of $2k+1$ leaf nodes each of m points. The i^{th} leaf node of this R-tree is a C_2 cluster if i is even and a C_1 cluster otherwise. Since the bounding box of any C_1/C_2 type cluster overlaps with the y-interval $[4,5]$ and is contained in the x-interval $[1,n]$, it follows that the range query $q=[1,n][4,5]$ retrieves all the leaf nodes. Since the number of such leaf nodes is n/m, where m is a constant, the time complexity for the query q is $O(n)$. □

A.2 Range Query Time Complexity of BANG files

We show that a BANG file has a worst-case range query time complexity of $O(n)$ for n 2-dimensional points. Our proof is similar to the one for the R-trees. We give a specific dataset for which all the leaf nodes (specified by their bounding rectangles) in the BANG file intersect a specific query q. Since, the BANG file [15] partitions the domain alternately, we choose each axis to be of length $D=[1,4^{nm}]$. We specify the set of points at different stages of the index construction. We start off with the following initial set of points.

$$I_0 = \{(1,1), (4^{nm}/2+1, 1), (1,3), (4^{nm}/2+1, 3)\}.$$

Using this set, the next set of points to be inserted is determined as:

$$NI_1 = \{(q_x, q_y) |\, \exists (p_x, p_y) \in I_0 : (q_x = p_x + 4^{nm}/2^2) \wedge ((q_y = 2) \vee (q_y = 4))\}.$$

Let the combined set be $I_1 = I_0 \cup NI_1$. Similarly, the insertions in the $(i+1)^{th}$ stage are:

$$NI_i = \{(q_x, q_y) |\, \exists (p_x, p_y) \in I_{i-1} : (q_x = p_x + 4^{nm}/2^{i+1}) \wedge ((q_y = 2) \vee (q_y = 4))\}.$$

For the above sequence of insertions, we can show that the data is always split using their x-coordinates. Consequently, all nodes that are formed intersect a query $q = [1, 4^{nm}][2, 2]$ leading to a time complexity of $O(n)$. As in the case of R-trees, every query that intersects a leaf node also intersects its ancestors, the query time complexity in terms of the node accesses is $O(n)$.

Theorem A.2 *In a BANG file for* n *2-dimensional points, the worst-case time complexity of range queries is* $O(n)$.

B Lower Bound on Range Query Time Complexity: Proofs

This appendix contains proofs leading to Lemma 1, which is used to obtain the lower bound on range query time complexity (Theorem 1).

The following two properties result from non-replication of data.

Lemma B.1 *For any vertex* v *in a non-replicating index, the point-query* $\Psi_d(p)$ *satisfies* $cond(v)$ *if* $p \in D(v)$.

Proof: Consider a point $p \in D(v)$. Since the index is non-replicating, point p is stored in only one vertex, say vertex u. In addition, $p \in D(v)$ implies vertex u is either the same as vertex v, or, a descendant of v. In either case, since the index is a tree, the path from the root to vertex u goes through vertex v. Next, by definition of a range query, $\Psi_d(p)$ retrieves point p. This implies $\Psi_d(p)$ satisfies the conditions associated with all the vertices on the path from the root to vertex u. Since v is on this path, $\Psi_d(p)$ satisfies $cond(v)$. □

Lemma B.2 *For any vertex* v *in a non-replicating index, if* $p = (p_1, p_2, \ldots, p_d) \in D(v)$, *then the interval* $[s_i, e_i]$ *along dimension* X_i, *where* $s_i \leq p_i \leq e_i$, *satisfies the sub-clause* C_i *of* $cond(v)$.

Proof: By contradiction. Consider a query $q = \langle [a_1, b_1], \ldots, [a_d, b_d] \rangle$, where $a_i = s_i$, $b_i = e_i$ and $a_j = b_j = p_j$ for all $j \neq i$. Then, by assumption, q does not satisfy the sub-clause C_i of $cond(v)$ and therefore $cond(v)$. Consequently, the point p, which falls in the query range of q is not retrieved by q. This contradicts the range query semantics. □

The following property holds for any subset S of S_d^n.

Lemma B.3 *For any set* $S \subseteq S_d$, $\Sigma_{1 \leq i \leq d} |P_i(S)| \geq d|S|^{1/d}$.

Proof: By induction on d.

Basis: Let S be a 1-dimensional point set. Since $P_1(S) = S$, the lemma trivially holds.

Hypothesis: Assume it holds for $d = l$.

Let $S \subseteq S_{l+1}$. Consider the l-dimensional set S' obtained by deleting the X_{l+1}-coordinate for each point $p \in S$, i.e.,

$$S' = \{(p_1, \ldots, p_l) \mid (p_1, \ldots, p_l, p_{l+1}) \in S\}.$$

Clearly, $S' \subseteq S_l$. Hence, by induction hypothesis,

$$\sum_{1 \leq i \leq l} |P_i(S')| \geq l|S'|^{1/l}. \tag{2}$$

The number of points in S is at most a product of those in S' and the number of distinct X_{l+1}-coordinates (given by $|P_{l+1}(S)|$). Hence,

$$|P_{l+1}(S)| * |S'| \geq |S|. \tag{3}$$

Since S' and S_{l+1} have the same projection-sets in the dimensions X_1 to X_l (i.e, $\forall i : (1 \leq i \leq l) : |P_i(S')| = |P_i(S)|$), the required summation can be computed as

$$\sum_{1 \leq i \leq l+1} |P_i(S)| = \sum_{1 \leq i \leq l} |P_i(S')| + P_{l+1}(S). \tag{4}$$

Using equations 2, 3 and 4, we obtain

$$\sum_{1 \leq i \leq l+1} |P_i(S)| \geq l * |S'|^{1/l} + \frac{|S|}{|S'|}.$$

The above sum is minimum when $|S'| = |S|^{l/(l+1)}$, the minimum value being $(l+1)|S|^{1/(l+1)}$. Hence the lemma holds. □

Lemma B.4 *Let C be a conjunction of binary comparisons, each involving an endpoint of an arbitrary 1-dimensional interval x and a constant. Let P be a subset of S_1 such that for each $p \in P$, $\Psi_1(p)$ satisfies C. Then, there are at least $(|P| - 1)$ point-queries corresponding to gaps in P that also satisfy C.*

Proof: If P has less than 2 elements, then the lemma is trivially true. Assume P has at least 2 elements. Let $C = C^s \wedge C^e$, where C^s involves only the left endpoint x^s and C^e involves only the right endpoint x^e.

Consider the conjunction of comparisons C^s. Each comparison is of the form (x^s *op* c), where *op* is one of the three relational operators, $<, >$, or $=$. The $\leq$ and $\geq$ operators can be rewritten using $<$ and $>$ respectively by modifying the constants appropriately. Since P has at least two elements, an equality comparison is not permissible in C^s. Let $C^s = C^s_g \wedge C^s_l$, where the comparisons in C^s_g involve only the *greater than* ($>$) operator and those in C^s_l involve only the *less than* ($<$) operator. The conjunction C^s_g can be reduced to one comparison between x^s and the maximum value of all the constants in its comparisons. C^s_l can be similarly reduced by taking the minimum value of its constants. Let $C^s = (x^s > c^s_1) \wedge (x^s < c^s_2)$. (If C^s does not have a comparison of the form $(x^s > c^s_1)$, then c^s_1 can be set to $-\infty$; similarly, if the comparison $(x^s < c^2_s)$ is missing, then c^s_2 can be set to ∞).

Now consider the set of points in P. Sort them in ascending order. Consider any pair of consecutive points a and b in this sorted sequence. Since P is a subset of S_1, a and b are odd values. Therefore, there exists a gap g such that $a < g < b$. Next, note that the point-queries for a and b satisfy the conjunct C and therefore, the conjuncts C^s and C^e. Consider the conjunct C^s. Since $\Psi_1(a)$ satisfies C^s, we have $(a > c^s_1) \wedge (a < c^s_2)$, i.e., $c^s_1 < a < c^s_2$. Likewise, for b we have $c^s_1 < b < c^s_2$. Since $a < g < b$, it follows that $c^s_1 < g < c^s_2$, i.e., $\Psi_1(g)$ satisfies C^s. A similar argument establishes that $\Psi_1(g)$ also satisfies C^e. Therefore, the query $\Psi_1(g)$ satisfies the conjunct C.

Next, let us count the number of gaps that satisfy C. The number of consecutive pairs in the sorted sequence of P is $(|P|-1)$. Since each such pair is separated by at least one gap, we have at least $(|P|-1)$ point-queries corresponding to gaps that satisfy the conjunct C. □

Next, we estimate the number of holes that satisfy $cond(v)$ for any vertex $v \in T$. We restrict ourselves to vertices that have at least one data point in their subtrees (i.e., $D(v) \neq \phi$).

Lemma B.5 *For any vertex $v \in T$, the number of hole queries that satisfy $cond(v)$ is at least $d(n_D(v)^{1/d} - 1)$.*

Proof: First, we estimate the number of i-hole queries that satisfy $cond(v)$. Each i-hole query corresponds to a gap in the i^{th} dimension and has an interval $[1, 2k+1]$ in all other dimensions. Now, consider $cond(v)$. Since it is expressed as $cond(v) = C_1 \wedge C_2 \wedge \ldots, C_d$, the i-hole query satisfies $cond(v)$ if its interval in the j^{th} dimension satisfies the conjunct C_j.

First, we show that, when $j \neq i$, the conjunct C_j is satisfied by the corresponding interval of every i-hole query. Consider any i-hole query. It has a query interval of $[1, 2k+1]$ in dimension X_j. Since $D(v) \neq \phi$, there exists a point $p \in D(v)$ and since $D(v) \subseteq S_d$, the X_j-coordinate of the point p, p_j, satisfies: $1 \leq p_j \leq 2k+1$. Then, from Lemma B.2 it follows that the sub-clause C_j is satisfied by an interval $[1, 2k+1]$, i.e., by the j^{th} interval of any i-hole query.

Given the above argument, the total number of i-holes that satisfy $cond(v)$ is given by the number of them satisfying the sub-clause C_i of $cond(v)$. To estimate this, we first note that an i-hole corresponds to a gap in $P_i(D(v))$, the i-projection set of $D(v)$. Hence, the number of i-holes satisfying $cond(v)$ is given by the number of gaps of $P_i(D(v))$ satisfying C_i. From Lemma B.4, this number is at least $(|P_i(D(v))| - 1)$. This gives a lower bound on the number of i-holes that satisfy $cond(v)$.

By summing the number of i-holes that satisfy the condition $cond(v)$ for all i, the total number of holes that satisfy $cond(v)$ is at least $\Sigma_{1 \leq i \leq d}(|P_i(D(v))| - 1)$. From Lemma B.3, this number is at least $d(|D(v)|^{1/d} - 1)$. □

Lemma B.6 *A vertex v in a non-replicating index structure for S_d^n is accessed by at least $(d((n_D(v))^{1/d} - 1)$ hole queries.*

Proof: From Lemma 1, there are at least $(d((n_D(v))^{1/d} - 1)$ hole queries that satisfy $cond(v)$. Next, we show that there are at least as many hole queries that access the vertex v.

Let u be the parent of v. Then, $D(v) \subseteq D(u)$. Hence for each dimension i, the gaps in $P_i(D(v))$ are also gaps in $P_i(D(u))$. Consider a specific gap in $P_i(D_v)$. This corresponds to the i^{th} interval of an i-hole query. From Lemma B.4, this interval satisfies the sub-clause C_i of $cond(u)$. Besides, the j^{th} interval of this i-hole query is $[1, 2k+1]$, which satisfies the sub-clause C_j of $cond(u)$ (by Lemma B.2). Hence, the i-hole query (corresponding to a gap in $P_i(D_v)$) satisfies $cond(u)$. Note that the same i-hole also satisfies $cond(v)$ (by Lemma B.4). Since there are at least $(d((n_D(v))^{1/d} - 1)$ such gaps in $P_i(D_v)$ (by Lemmas B.4 and B.3), we have that at least $(d((n_D(v))^{1/d} - 1)$ i-hole queries satisfy both $cond(u)$ and $cond(v)$. Extending this argument till the root, there exist at least $(d((n_D(v))^{1/d} - 1)$ hole queries that satisfy all the conditions on the path from the root to vertex v. Consequently, from the query algorithm, it follows that at least as many hole queries access vertex v. $\Box$

Lemma 1 *Let T be a non-replicating index for the set of points S_d^n. Then, the total access cost for all the hole queries in T is at least $n - o(n)$.*

Proof: We prove this for any subset $S \subseteq S_d^n$ using induction on the height h of the tree T for S.

Basis: h=0. Then, the tree consists of one single node – the root with all the data in it. If $n \geq 2$, there exists at least one hole that searches in the data items associated with the root. Hence the cost is n.

Hypothesis: Assume it is true for all trees of height $\leq l$.

Consider a tree T of height $l + 1$. Then, the total access cost for all the hole queries is given by the cost at the root followed by the cost at the subtrees corresponding to its children. Each hole query that accesses the root searches all the data items incurring a cost of $(1 + |data(R)|)$, the initial unit cost being for accessing the root. The number of such hole queries is obtained from Lemma B.6. Hence the total access cost for all hole queries on tree T rooted at R (denoted by $c(R)$) is:

$$c(R) = d(n_D(R)^{1/d} - 1)(1 + |data(R)|) + \sum_{v \in children(R)} c(v).$$

Since $n_D(R) = n$, we have

$$c(R) = d(n^{1/d} - 1)(1 + |data(R)|) + \sum_{v \in children(R)} c(v).$$

Note that each subtree rooted at any child of R is of height at most l. Then, by induction hypothesis, we have for each child v of R, $c(v) \geq n_D(v) - o(n_D(v))$. Since $\Sigma_{v \in children(R)} n_D(v) = n - |data(R)|$, we have

$$c(R) = d(n^{1/d} - 1)(1 + |data(R)|) + (n - |data(R)|) - \sum_{v \in children(R)} o(n_D(v)).$$

Since the root has at most a constant number, r, of children, we have

$$c(R) \geq d(n^{1/d} - 1)(1 + |data(R)|) + (n - |data(R)|) - r * o(n) \geq n - o(n).$$

Hence, for all trees T the total access cost is at least $n - o(n)$. $\Box$

C Complexity Analysis of An O-tree: Proofs

C.1 Query Time Complexity Analysis of An O-tree: Proofs

Lemma 2 *In a B-tree for a set of partitions using σ-order, at most 2 partitions that intersect a query are not contained in the query in dimension σ.*

Proof: Assume for contradiction that more than two, say three, bounding rectangles are not contained in the query in dimension σ. Let $[f, g]$, $[l, m]$ and $[u, v]$ be the intervals of these bounding rectangles in dimension σ and $[s, t]$ be the query interval in dimension σ. Then, since $[f, g]$ is not contained in $[s, t]$, we have $\neg((s \leq f) \wedge (g \leq t))$, i.e., $(f < s) \vee (t < g)$. Since $[f, g]$ intersects $[s, t]$, we have either $(f < s \leq g)$ or $(f \leq t < g)$. Similarly, for the other two, we have $(l < s \leq m)$ or $(l \leq t < m)$, and, $(u < s \leq v)$ or $(u \leq t < v)$. Of these three, two of them involving the same query endpoint have to be true. Without loss of generality, let them be $(f < s \leq g)$ and $(l < s \leq m)$. This implies that one of the following 4 combinations holds – (1) $f \leq l < g \leq m$, (2) $l \leq f < m \leq g$, (3) $f \leq l < m \leq g$, or (4) $l \leq f < g \leq m$. However, since $[f, g]$ and $[l, m]$ are the intervals of two distinct entries, from Lemma C.1 one of $f \leq g \leq l \leq m$ and $l \leq m \leq f \leq g$ also holds, leading to a contradiction. □

Lemma C.1 *Let $E^1, \ldots, E^l$ denote the sequence of entries at any level of a B-tree for a set of partitions using σ-order. Then, for any two entries E^i, E^j, $i < j$, the intervals b^i_σ, b^j_σ of the corresponding bounding rectangles satisfy the following relation. If $b^i_\sigma = [l, m]$ and $b^j_\sigma = [s, t]$, then $l \leq m \leq s \leq t$.*

Proof: Consider any two entries E^i and E^j. Since the B-tree is built using the partition extents, the associated extents e^i and e^j are ordered as $e^i \prec_\sigma e^j$. Let $e^i = (p^i_1, p^i_2)$ and $e_j = (p^j_1, p^j_2)$. Since $e^i \prec_\sigma e^j$, we have $p^i_1 \prec_\sigma p^i_2 \prec_\sigma p^j_1 \prec_\sigma p^j_2$. Let σ^s_r denote the σ-coordinate of a point p^s_r. Then, from the above equation and from the definition of the $\prec_\sigma$ relation, we infer that $\sigma^i_1 \leq \sigma^i_2 \leq \sigma^j_1 \leq \sigma^j_2$. By definition of a bounding rectangle, interval $b^i_\sigma = [\sigma^i_1, \sigma^i_2]$ and interval $b^j_\sigma = [\sigma^j_1, \sigma^j_2]$. Hence, the relation holds. □

Lemma C.2 *In a d-layered B-tree of k partitions, at most $O(k^{(d-1)})$ leaf partitions that intersect a query are not contained in it. The overhead of identifying these leaf partitions is $O(k^{(d-1)} \log k)$.*

Proof: The complexity of a query q on a d-layered B-tree of k partitions can be analyzed using the *SearchDtree(q, D, i)* algorithm given below. Given the lower i-layers of the d-layered B-tree, a query q identifies two sets of leaf partitions – VS_0 and VS_1. The set VS_0 is the set of leaf partitions that are *contained* in q, whereas the set VS_1 is the set of leaf partitions that intersect q but are not contained in it in at least one dimension. These two sets are obtained as follows. First, the layer-i B-tree is searched using the query. From Lemma 2, this search yields at most two leaves L_1, L_m that intersect q but are not contained in it in dimension X_1. These leaves correspond to $(i-1)$-layered D-trees. Therefore, all leaf partitions, emanating from these two vertices and intersect the query q, are not contained in query in at least one dimension. In the worst-case, all leaf partitions of these two vertices satisfy this property and therefore, are included in VS_1 in Step 12. For the interior vertices $L_2, \ldots, L_m$, the query searches recursively in the next $(i-1)$-layers of the D-tree. When $i = 1$, the leaves $L_2, \ldots, L_{m-1}$ correspond to partitions that are contained in the query in all dimensions. Hence they are included in VS_0.

function *SearchLayeredBtree(q, D, i)*: $\langle$ Two sets of vertices $\rangle$

begin

(1) $\langle L_1, L_2, \ldots, L_m \rangle :=$ *SearchBTree(q, D)*;

(2) **If** $(d = 1)$

then

begin

(3) $VS_0 := \{L_2, \ldots, L_{m-1}\}$;

(4) $VS_1 := \{L_1, L_m\}$;

end

else

begin

(5) $VS_0 := \phi$;

(6) $VS_1 := \phi$;

(7) **foreach**$L \in \{L_2, \ldots, L_{m-1}\}$ **do**

begin

(8) $\langle TVS_0, TVS_1 \rangle :=$ *SearchLayeredBtree(*$q, L, i-1$*)*;

(9) $VS_0 := VS_0 \cup TVS_0$;

(10) $VS_1 := VS_1 \cup TVS_1$;

end

(11) **foreach**$L \in \{L_1, L_m\}$ **do**

(12) $VS_1 := VS_1 \cup$ *RetrieveAllLeafPartitions(*$L, i-1$*)*;

end

(13) **return** $\langle VS_0, VS_1 \rangle$

end

Now, we estimate the cardinality of the sets VS_0 and VS_1 for a d-layered B-tree of k partitions. Let these be denoted by $N_0(d, k)$ and $N_1(d, k)$ respectively, where k is the number of partitions in the ordered tree at each layer. Then, from the above algorithm it follows that:

$$N_1(d, k) = 2[N_0(d-1, k) + N_1(d-1, k)] + (m-2)N_1(d-1, k);\ 2 \leq m \leq k.$$

$$N_0(d, k) = (m-2)N_0(d-1, k);\ 2 \leq m \leq k.$$

$$N_1(1, k) = 2;\ \ N_0(1, k) = (m-2); 2 \leq m \leq k.$$

Solving these two equations, we obtain that $|VS_0| = N_0(d) = O(k^d)$ and $|VS_1| = N_1(d) \leq 2k^{d-1} = O(k^{d-1})$. Note that of the two sets, VS_0 and VS_1, only VS_1 contributes to query overheads. (All points in the partitions of the set VS_0 are part of the query result and hence do not contribute to the query overheads). Let us now estimate these overheads of locating the partitions in the set VS_1. Let $QD(d, k)$ denote this complexity, where d denotes the number of dimensions in which the query may not contain the partitions. From Lemma C.1, we have $QD(1, k) = \log k$. Otherwise,

$$QD(d, k) = 2k^{d-1} + (m-2)QD(d-1, k); 2 \leq m \leq k$$

Solving this equation, we obtain $QD(d, k) = O(k^{(d-1)} \log k)$. $\square$

Theorem 4 *Range queries in an O-tree of n d-dimensional points can be answered in $O(n^{(d-1)/d} + k)$ time, where k is the number of retrieved data points. Exact match queries can be answered in $O(\log n)$ time.*

Proof: First, let us examine range query time complexity. The range query overheads in an O-tree can be analyzed using the following algorithm.

1. Search in first-level d-layered B-tree. Identify the set of intersecting leaf partitions in this tree. Of these, only those leaf partitions that are not contained in the query contribute to the overheads. Let P_1 denote the set of such partitions.

2. For each partition in P_1, search in the corresponding second-level d-layered B-trees. Identify the set of intersecting partitions of the second-level D-trees. Let P_2 denote the set of partitions that are not contained in the query.

3. For each partition in P_2, search in the corresponding static k-d tree.

Next, let us analyze the complexity of each of these steps in turn.

1. The first-level is a d-layered tree of at most $k = 4n^{1/d}$ partitions. Therefore, from Lemma C.2, the first step has $O(n^{(d-1)/d} \log^{d-2} n)$ overheads. The cardinality of the set P_1 is $O(n^{(d-1)/d} \log^{d-1} n)$.

2. Step 2 is executed $|P_1|$ times. At the second-level, we have d-layered B-tree of at most $k = 4 \log n / \log \log n$ partitions. Therefore, the total overhead for all the $|P_1|$ searches is $O(n^{(d-1)/d} / \log^{d-1} n) * O(\log^{(d-1)/dn} / \log^{d-2} \log^d n)$ $= O(n^{(d-1)/d})$. Also, the cardinality of set $P_2 = O(n^{(d-1)/d} / \log^{d-1} n) * O(\log^{d-1} n / \log^{d-1} \log^d n)$ $= O(n^{(d-1)/d} / \log^{d-1} \log^d n)$.

3. Step 3 is executed $|P_3|$ times. Since the static k-d trees have $O(\log^d \log^d n)$ points, the query time overhead from theorem 2 is $O(\log^{d-1} \log^d n)$. Thus, the total query overhead for Step 3 is $O(n^{(d-1)/d})$.

Combining the above query overheads for the three levels of the combined k-d tree yields $O(n^{(d-1)/d})$ range query complexity.

Next, let us examine exact match query overheads. Starting from the layer-$2d$ ordered tree, we proceed down to a static k-d tree, identifying the partition in which the query fits in (according to the algorithm of Section 3.2.2). Since at most one partition is identified at each layer, the search is a single path from the root to the static k-d tree and takes $O(\log n)$ time at each layer. Since exact queries can be answered in $O(\log n)$ time in the static k-d trees, the total time taken is $O(\log n)$. $\Box$

C.2 Insertion/Deletion Time Complexity Analysis of An O-tree: Proofs

Lemma 3 *An O-tree of n points always exhibits the following characteristics.*

1. *At layer 0, the number of points in any static k-d tree is at most $2^{d+1} \log^d \log^d n$.*

2. *The number of partitions in a layer-i B-tree is at most*

   (a) $4 * n^{1/d}/\log n$ *if* $d+1 \leq i \leq 2d$,

   (b) $4 * \log n/\log\log^d n$ *if* $1 \leq i \leq d$.

Proof: Assume the whole structure was last constructed for n_0 points. Since the number of updates (insertions/deletions) that can occur before the next reconstruction is $\alpha_{2d}n_0$, the current number of points n in the tree satisfies the relation: $n \geq (1-\alpha_{2d})n_0$ (i.e., $n_0 \leq n/(1-\alpha_{2d})$). We use this observation to prove the lemma.

Consider the static k-d tree. It has at most $(1+\alpha_0)s_0(n_0) = 2s_0(n_0)$ points. Once this limit is exceeded, it is split into two trees. Therefore, the maximum number of points is given by

$$2s_0(n_0) = 2\log^d\log^d n_0 \leq 2\log^d\log^d(n/(1-\alpha_{2d})) < 2(d\log\log(n+\frac{1}{2}))^d < 2(d\log(2\log n))^d < 2^{d+1}\log^d\log^d n.$$

Consider a layer-i B-tree B, $1 \leq i \leq 2d$. Just after construction of the O-tree, the layer-i B-tree has $s_i(n_0)/s_{i-1}(n_0)$ partitions. At most $\alpha_i s_i(n_0)$ updates can occur in this tree. Since each partition (the corresponding $(i-1)$-layered tree) is reconstructed after $\alpha_{i-1}s_{i-1}(n_0)$ updates occur in it, the number of new partitions in B due to $\alpha_i s_i(n_0)$ updates is at most $(\alpha_i s_i(n_0))/(\alpha_{i-1}s_{i-1}(n_0))$. Hence the total number of partitions in B is at most

$$L_i = (1+\frac{\alpha_i}{\alpha_{i-1}})\frac{s_i(n_0)}{s_{i-1}(n_0)} < 2\frac{s_i(n_0)}{s_{i-1}(n_0)}.$$

Consider B-trees in the top d layers, i.e., $d+1 \leq i \leq 2d$.

$$L_i < 2\frac{s_i(n_0)}{s_{i-1}(n_0)} = 2\frac{(n_0)^{1/d}}{\log n_0} < 2\frac{n_0^{1/d}}{\log n_0 - \log(1-\alpha_{2d})}.$$

Since $\alpha_{2d} < 1 - \frac{1}{\sqrt{2}}$,

$$L_i < 2\frac{(n_0)^{1/d}}{\log n_0 - \log(1-\alpha_{2d})} < 2\frac{(n/(1-\alpha_{2d}))^{1/d}}{\log n_0} < 2 * 2^{1/2d}\frac{n^{1/d}}{\log n} < 4\frac{n^{1/d}}{\log n}.$$

Consider B-trees in the next d layers, i.e., $1 \leq i \leq d$.

$$L_i < 2\frac{s_i(n_0)}{s_{i-1}(n_0)} = 2\frac{\log n_0}{\log\log^d n_0} = 2\frac{\log n_0}{\log(\log n + \frac{1}{2})^d} < 2\frac{\log n_0}{\log\log^d n)} < 2\frac{\log n + 1/2}{\log\log^d n} < 2\frac{\log n}{\log\log^d n}.$$

Hence the lemma holds. □

Theorem 6 *Amortized insertion/deletion cost in an O-tree of n data points is $O(\log n)$* .

Proof: The cost of searching in $2d$ B-trees, one each at the first 2d-layers is $O(\log n)$. The insertion/deletion cost in a static k-d tree is $O((\log^d\log^d n) * \log(\log^d\log^d n)) = O(\log n)$. Hence, the total cost when there are no reorganizations in the first 2d-layers is $O(\log n)$.

Next, let us examine the reorganization costs. Assume the O-tree was last constructed for n_0 points. Each i-layered O-tree with $s_i(n_0)$ data points is reorganized only after $\alpha_i s_i(n_0)$ insertions or deletions have occurred in it. Consequently, the size can increase to at most $(1+\alpha_i)s_i(n_0)$ points. These points are split into two partitions and an i-layered O-tree is constructed for each of the partitions. From Theorem 5, the construction cost is $O(s_i(n_0)\log(s_i(n_0)))$. Since $\alpha_i s_i(n_0)$ updates occurred before such a reconstruction, the amortized cost, which is $O(\log s_i(n_0)) = O(\log n)$, is charged to each of these updates. Consequently, each update accrues an amortized cost of $O(\log n)$ due to reconstruction at each layer. Since there are a constant number, $2d + 1$, of such layers, the total amortized cost is $O(\log n)$. Hence the result follows. $\square$

D Conversion of Amortized Insertion/Deletion Bounds of An O-tree to Worst-case Bounds

As in the divided k-d tree, the amortized times for insertions and deletions of Theorem 6 can be changed into worst-case bounds using the general technique of global rebuilding described in [31]. This technique is based on the following idea. Consider a structure S that supports *weak* insertions and *weak* deletions. Insertions and deletions on a structure of n points are *weak*, if after αn insertions and/or deletions, with $0 < \alpha < 1$, all time and storage bounds of the structure have not increased asymptotically. The global rebuilding technique converts weak updates (insertions and deletions) into normal worst-case updates as follows: Maintain two copies of the data structure – the first one is constructed for n points and the second one is constructed after an additional $\alpha n/2$ updates have occurred. Let the number of points in the structure at this time be n_1. Queries are processed on the first structure till at most αn updates occur on it. Then, they switch to the second structure, the construction of which is completed by that time. Hence, query processing time is unaffected. Next, let us see how the second structure is constructed. Instead of constructing the second structure all at once, the construction work is distributed across $\alpha n_1/6$ updates that follow. Since the construction can be accomplished in $O(n_1 \log n_1)$ time, this strategy adds a logarithmic overhead to each of the αn_1 updates. The updates that occur on the first structure after the construction of the second has started are buffered so that they can be applied on the second structure once its construction is completed. By ensuring that the number of such buffered updates is less than $\alpha n_1/2$, we have at most one structure under construction at all times. This technique is called global rebuilding and is explained in more detail in [24]. The following theorem summarizes this technique.

Theorem (Global Rebuilding) *Given a tree T representing n points with rebuilding time bounded by $O(n \log n)$ that allows for weak insertions and weak deletions, we can dynamize it into a structure T' such that*

- *the insertion time on T' is $O(WI_T(n) + \log n)$;*
- *the deletion time on T' is $O(WD_T(n) + \log n)$;*
- *the query time and amount of storage space do not change asymptotically.*

Here, $WI_T(n)$ and $WD_T(n)$ denote the worst-case times for performing weak insertions and weak deletions on a structure T of n points. An insertion (or deletion) on a structure T is called *weak* [24] if there exists an α,

$0 < \alpha < 1$, such that after αn insertions (or deletions), all time and storage bounds of the structure have not increased asymptotically.

We apply this theorem on each i-layered O-tree starting from $i = 1$ to $i = 2d$ in succession. This process converts all amortized bounds on these structures to worst-case bounds. Hence, the following result holds.

Theorem D.1 *An O-tree of n points can support insertions and deletions in $O(\log n)$ worst-case time.*

Proof: By applying global rebuilding theorem, we prove that each i-layered O-tree, $0 \leq i \leq 2d$, can support insertions/deletions in $O(\log n)$ worst-case time. Our proof is using induction on i.
Basis: $i = 0$. The tree corresponds to the layer-0 static k-d tree. It has $O(\log^2 \log^2 n)$ points and is reconstructed after each insertion/deletion, the cost being $O(\log^2 \log^2 n \log \log^2 \log^2 n) = O(\log n)$.
Hypothesis: Assume it is true for all layers $i \leq l - 1$. Consider a layer-l O-tree. This structure is reconstructed only after $\alpha_l s_l(n_0)$ updates occur in it. Each of these updates incurs a worst-case time of $O(\log n)$ in the layer-l B-tree and $O(\log n)$ in the layer-$(l-1)$ O-tree (by induction hypothesis). Hence, the layer-l O-tree supports weak updates. Applying the global rebuilding theorem, these insertion and deletion times are converted into worst-case times. Hence the theorem holds. $\square$

References

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger.
The R* tree: An efficient and robust access method for points and rectangles.
Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, pages 322–331, May 23-25 1990.

[2] J. L. Bentley.
Multi-dimensional binary search trees used for associative searching.
Communications of the ACM, 18:509–517, 1975.

[3] J. L. Bentley and H. A. Maurer.
Efficient worst-case data structures for range searching.
Acta Informatica, pages 155–168, 1980.

[4] S. Berchtold, D. A. Keim, and H. P. Kreigel.
The X-tree: An index structure for high dimensional data.
Proc. of the Intl. Conf. on Very Large Databases, 1996.

[5] B. Chazelle.
Filtering search: A new approach to query-answering.
Proc. IEEE Symp. on Foundataions of Computer Science, 24:122–132, 1983.

[6] B. Chazelle.
Lower bounds on the complexity of multi-dimensional searching.
Proc. IEEE Symp. on Foundataions of Computer Science, pages 87–96, 1986.

[7] B. Chazelle.
Lower bounds for orthogonal range searching: I. The reporting case.
Journal of the ACM, 37(2):200–212, April 1990.

[8] B. Chazelle.

Lower bounds for orthogonal range searching: II. The arithmetic model.

Journal of the ACM, 37(3):439–463, July 1990.

[9] B. Chazelle.

Lower bounds for off-line range searching.

Proc. ACM Symp. on Theory of Computing, pages 733–740, 1995.

[10] Y. J. Chiang and R. Tamassia.

Dynamic algorithms in computational geometry.

Proc. of the IEEE, 80(9):1412–1433, September 1992.

[11] ESRI Inc.

Spatial Data Engine.

Manual, 1996.

[12] M. L. Fredman.

A lower bound on the complexity of orthogonal range queries.

Journal of the ACM, 28:696–705, 1981.

[13] M. L. Fredman.

Lower bounds on the complexity of some optimal data structures.

SIAM Journal of Computing, 10:1–10, 1981.

[14] M. L. Fredman.

The inherent complexity of dynamic data structures which accomodate range queries.

Proc. IEEE Symp. on Foundataions of Computer Science, 21:112–121, 1983.

[15] M. W. Freeston.

The BANG file: A new kind of grid file.

Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, pages 260–269, May 1987.

[16] M. W. Freeston.

A general solution of the n-dimensional B-tree problem.

Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, May 1995.

[17] A. Guttman.

R-trees: A dynamic index structure for spatial searching.

Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, pages 47–57, 1984.

[18] J. M. Hellerstein, J. M. Koutsoupias, and C. H. Papadimitrou.

On the analysis of indexing schemes.

Proc. of the ACM Priniciples of Database Systems, May 1997.

[19] D. B. Lomet and B. Salzberg.

The hB-tree: A multi-attribute indexing method with good guaranteed performance.

ACM Transactions on Database Systems, 15(4):625–658, December 1990.

[20] K. Mehlhorn.

Data Structures and Algorithms 3: Multidimensional Searching and Computational Geometry.

Springer-Verlag, 1984.

[21] J. Nievergelt, H. Hinterberger, and K. C. Sevcik.

The gird file: An adaptable, symmetric multi-key file structure.

ACM Trenasactions on Database Systems, 9(1):38–71, March 1984.

[22] M. H. Nodine, M. T. Goodrich, and J. S. Vitter.

Blocking for external graph searching.

Algorithmica, 16(2):181–214, 1996.

[23] Oracle Inc.

Spatial Data Option from Oracle.

Manual, 1996.

[24] M. Overmars.

The design of dynamic data structures.

Lecture Notes in Computer Science 156, 1983.

[25] M. Overmars and J. van Leeuwen.

Worst-case optimal insertion and deletion methods for decomposable searching problems.

Information Processing Letters, 12(4):168–172, 1981.

[26] M. Overmars and J. van Leeuwen.

Dynamic multi-dimensional data structures based on quad and k-d trees.

Acta Informatica, 17:267–283, 1982.

[27] H. Samet.

The design and analysis of spatial data structures.

Addison-Wesley, 1989.

[28] T. Sellis, N. Roussopoulos, and C. Faloutsos.

The R+ tree: A dynamic index for multi-dimensional objects.

Proc. of the Intl. Conf. on Very Large Databases, pages 507–518, 1988.

[29] S. Subramanian and S. Ramaswamy.

The P-range tree: A new data structure for range searching in secondary memory.

Proc. ACM-SIAM Symposium on Discrete Algorithms, pages 378–387, 1995.

[30] P. M. Vaidya.

Space-time tradeoffs for orthogonal range queries.

SIAM Journal of Computing, 18(4):748–758, 1989.

[31] M. van Kreveld and M. Overmars.

The divided k-d tree.

Algorithmica, 6:840–858, 1991.

[32] A. C. Yao.

Space-time trade-off for answering range queries.

Proc. ACM Symp. on Theory of Computing, 14:128–136, 1982.