Asian Journal of Information Technology 15 (11): 1723-1730, 2016
ISSN: 1682-3915

An Optimal Algorithm for Range Search on Multidimensional Points

T. Hema and K.S. Easwarakumar
Department of Computer Science and Engineering, Anna University, 600025 Chennai, India

Abstract. This study proposes an efficient and novel method to address range search on multidimensional points in $\theta(t)$ time where t is the number of points reported in R^k space. This is accomplished by introducing a new data structure, called BITS kd-tree. This structure also supports fast updation that takes $\theta(1)$ time for insertion and O(log n) time for deletion. The earlier best known algorithm for this problem is $O(\log^k n+t)$ time in the pointer machine model.

Keywords: BITS kd-tree, threaded trie, range search, algorithm, time

INTRODUCTION

The kd-trees introduced by Bentley (1975, 1979) are multidimensional binary search trees commonly used for storing k dimensional points. They are also used to perform search operations such as exact match, partial match and range queries. Range queries are mostly used in GIS applications to locate cities within a certain region in a map. Similarly, in the geometrical view of a database, one can use orthogonal range search to perform a query. Generally, kd-trees with n nodes have a height n and hence the complexity for insertion and search are high. Although many multi-dimensional search structures are found in the literature (De Berg *et al*., 2008; Preparata and Shamos, 1985; Samet, 1974; 1990), they differ from the standard kd-trees mainly in the space-partitioning methods used. Recall that a 2-d tree stores two-dimensional point data of the form (x, y). A 2-d tree splits primarily on the x coordinate of a point at even level and then on the corresponding y coordinate at the odd level and so on. Hence, the trees are unbalanced and are not efficient for search operations. Also, the worst case time complexity for range search on a 2-d tree is $O(\sqrt{n}+t)$ where t is the number of points reported and for k dimensions it is $O(n^{1-1/k}+t)$ (Bentley, 1975; Lee and Wong, 1977). In general, most of the kd-tree variants get unbalanced when the data is clustered thereby affecting query operations. The PK k-d tree, Bucket PR k-d tree (Orienstein, 1982), PR k-d trees (Nelson and Samet, 1986) and Path level compressed PMR k-d trees (Nilsson and Tikkanen, 2002) are some of the trie-based kd trees used to store point data. However, these trees are not always balanced, especially when the data is clustered. One of the dynamic versions of k-d tree is the divided k-d trees (Kreveld and Overmars, 1991) for which the range query time is $O(n^{1-1/k} \log^{1-1/k}+t)$.

The best known dynamically balanced tree uses bitwise interlaced data (Tropf and Herzog, 1981) over kd-trees mapping k dimensions to one dimension. Although their search time is O(k(log n+t)) for reporting t points, bitwise interlacing leads to discarded areas during range search. In the case of squarish, kd-trees (Devroye *et al*., 2000), an x, y discriminant is based on the longest side of rectangle enclosing the problem space instead of alternating the keys. Recently, hybrid versions of squarish kd-tree, relaxed kd-tree and median kd-trees (Crespo, 2010) have overcome the problem of height balancing. An amortized worst case efficiency of range search for the hybrid squarish kd-trees, relaxed and median trees for k-dimensional partial match queries are $1.38628 \log_2 n$ and $1.38629 \log_2 n$ and $1.25766 \log_2 n$ respectively. Their experimental results match the aforementioned theoretical results, where they show that the hybrid median trees outperform the other variants. However, as far as query handling is concerned, these structures perform only partial match queries for two dimensions efficiently. The most recent work in the pointer machine model is an orthogonal range reporting data structure with $O(n(\log n/\log\log n)^d)$ space that address range queries in $O(n(\log n/\log\log n)^{d-4+1/(d-2)}+t)$ time where $d\geq 4$ (Afshani *et al*., 2012).

Range trees of (Bentley and Sax, 1979; Bentley, 1979) are yet another class of balanced binary search trees used for rectangular range search which showed improvement in the query time of $O(\log^k n+t)$ over $O(n^{1-1/k}+t)$ of k d-trees where k is the dimension for a set of n points and

Corresponding Author: T. Hema, Department of Computer Science and Engineering, Anna University, 600025 Chennai, India

t is the number of reported points. This was later improved to $O(\log^{k-1} n+t)$ using fractional cascading in layered range trees (Willard, 1978) but the space requirements are relatively high of $O(n \log^{k-1} n+t)$. A kd-Range DSL-tree performs k-dimensional range search in $O(\log^{k} n+t)$ time was proposed by Lamoureux and Nickerson (1995).

Recently, Chan *et al*. (2011) have proposed two data structures for 2d orthogonal range search in the word RAM model. The first structure takes $O(n \lg \lg n)$ space and $O(\lg \lg n)$ query time. They show improved performance over previous results (Alstrup *et al*., 2000) of which $O(n\lg^{\varepsilon} n)$ space and $O(\lg \lg n)$ query time or with $O(n \lg \lg n)$ space and $O(\lg^2 \lg n)$ query time. The second data strucure is based on $O(n)$ space and answers queries in $O(\lg^{\varepsilon} n)$ time that outperforms previous $O(n)$ space data structure (Nekrich, 2009), answers queries in $O(\lg n/\lg \lg n)$ time.

Furthermore, they also propose an efficient data structure for 3-d orthogonal range reporting with $O(n \lg^{1+\varepsilon} n)$ space and $O(\lg^2 \lg n+k)$ query time for points in rank space where $\varepsilon>0$. This improves their previous results (Chan *et al*., 2011) with $O(n \lg^2 n)$ space and $O(\lg \lg n+k)$ query time or with $O(n \lg^{1+\varepsilon} n)$ space and $O(\lg^2 \lg n+k)$ query time where k points are reported. Finally they have extended range search to higher dimensions also.

Since, such range queries are common among multi-dimensional queries in database applications, we have mainly considered an orthogonal range search on multi-dimensional points.

MATERIALS AND METHODS

Our contributions: In this research, we make use of the BITS-tree (Easwarakumar and Hema, 2013), a segment tree variant that performs stabbing and range queries on segments efficiently in logarithmic time. Most importantly, the distribution of the data points (uniform or skewed) does not affect the height of the BITS-tree and in turn facilitates faster search time. Here, we actually use the BITS-tree structure to store points related to each dimension and thereby form a multi-level tree, called BITS kd-tree. In addition, certain nodes of the BITS-tree associate to a variant of the trie data structure called threaded trie, to facilitate fetching a required node in constant time. Unlike, k-d trees, it does not associate co-ordinate axis, level wise, for comparison to locate or insert a point. Instead, the tree at the first level has nodes with a key on only distinct values of first co-ordinate of the points. Therefore, this tree corresponds to one dimensional data. This tree is then augmented with another tree at second level and there in key values of the nodes associated with distinct first two co-ordinates of the points. In general, ith tree corresponds to the distinct first i co-ordinates of the set of points given. Moreover, in each tree, the inorder sequence provides the sorted sequence. That is, BITS k-d trees is a multi-level tree and its construction is illustrated in the subsequent studies. BITS-trees to be removed.

Originally, the BITS-tree (Balanced Inorder Threaded Segment Tree) (Easwarakumar and Hema, 2015) is a dynamic structure that stores segments and also answers both stabbing and range queries efficiently. Unlike segment trees, it also permits insertion of segment with any interval range.

Definition 1: A BITS- tree is a height balanced two-way inorder-threaded binary tree T that satisfies the following properties:

- Each node v of T is represented as v([a,b], L) where [a,b] is the range associated with the node v and L is the list of segments containing the range [a,b], i.e., if $[c,d]\in L$ then $[a,b]\subseteq[c,d]$.
- Given v_i ([a,b], L_1) then v_i ([a,b], L_1)$\neq v_2$ ([a_2,b_2]) then:

$$[a_1,b_1]\cap[a_2,b_2]=\begin{cases}[b_1]\text{ if } a_2=b_1\\ [b_2]\text{ if } a_1=b_2\\ \varphi \text{ otherwise}\end{cases}$$

  i.e., ranges can either overlap only at end points or do not overlap at all.
- Suppose v_i ([a,b], L_1) appears before v_n ([a_n,b_n]) in the inorder sequence, then $b_1\leq a_2$
- It has a special node, called dummy node denoted by D with range and list as ϕ(empty)
- Suppose v_i ([a,b], L_1) and v_2 ([a_2,b_2]) are the first and last nodes of the inorder sequence respectively, then InPred(v_1) = In Succ(v_n)and the range, say [a,b] of any node contained in [a_1,b_n], i.e, $[a,b]\subseteq[a_1,b_n]$

Here, the functions InPred() and InSucc(), respectively returns inorder predecessor and successor. A sample BITS-tree is shown in Fig. 1. Note that the dangling threads actually point to a dummy node which is not shown in Fig 1. The BITS-tree is originally developed for storing segments, but we use this for a different purpose of storing points. Thus, we modify this structure to suit our requirement as described.

Each node v([a,b], L) is replaced by v(p, L′, T) where p is a point in R^k, $k\geq 1$and L′ is a pointer to the list of collinear points in dimension k+1 having p for the first k co-ordinates. However, this list is maintained in the tree at the next level which is described in the following

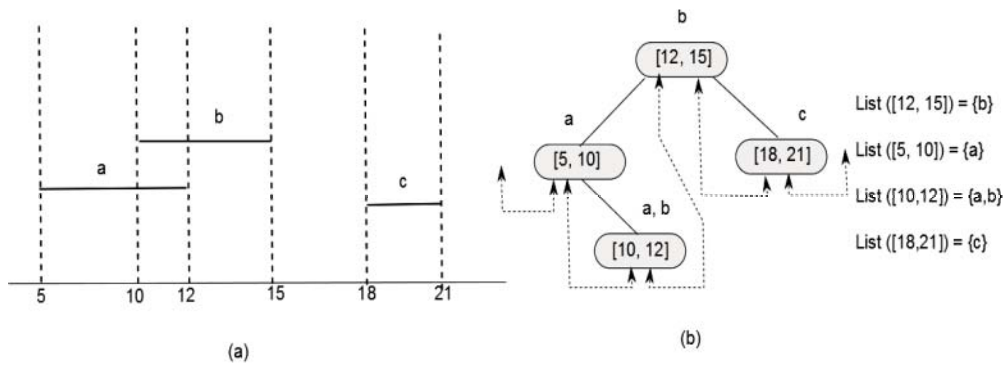

Fig. 1: a) Set of segments; b) BITS-tree for the given segments

subsection. Now, T is either null or a pointer to a threaded trie which is elaborated in the next section. For any two points p_1 and p_2 stored in a tree,$p_1 \neq p_2$. Suppose $v_1(p_1, L_1, T_1)$ appears before $v_1(p_2, L_2, T_2)$ in the inorder sequence, then $p_1<p_2$ as per the following definition.

Definition 2: Let, $p_1 = (x_1^1, x_2^1, \ldots x_k^1)$ and $p_2 = (x_1^2, x_2^2, \ldots x_k^2)$ be two points in a k-dimensional space then:

- $p_1 = p_2$ implies $x_j^1 = x_j^2$ for each j = 1, 2,... k
- p_1<(or)> p_2 implies head(p_1, j) = head(p_2, j) and x^1_{j+1} <(or >) x^2_{j+1} for some j

In the subsequent sections, for better clarity, we use hyphen(-) for a certain parameter of a node to denote that the particular parameter is irrelevant with respect to the context. For instance, (p, -, T) denotes that the list contents are irrelevant for that point p at this time.

Threaded trie: Threaded tries are variants of tries that consists of two types of nodes viz. trie node and data node. For instance, in Fig. 2, A-C are trie nodes and the rest are data nodes. Unlike in tries, the trie node here does not have a field for blank (b). However, each of these trie nodes contain two segments. One is the index pointer and the other is a tag value which is either 0 or 1 where 0 denotes the corresponding index point in a thread and otherwise it will be 1. Here, all null pointers are replaced by threaded pointers which point to the next valid node, if one exists. For instance, the thread pointers of 1-3 of node A points to the node C as this is the next valid node. Similarly, thread pointers of 0 and 1, in C points to the data node 42. Note here that ordering on the nodes provides the sorted sequence. Also, data nodes appear at the same level. This is accomplished by having uniform width for all data. For instance, the data 8 is treated as 08 in Fig. 2.

Construction of multi-level BITS-tree: Multi-level BITS-trees are constructed using a collection of

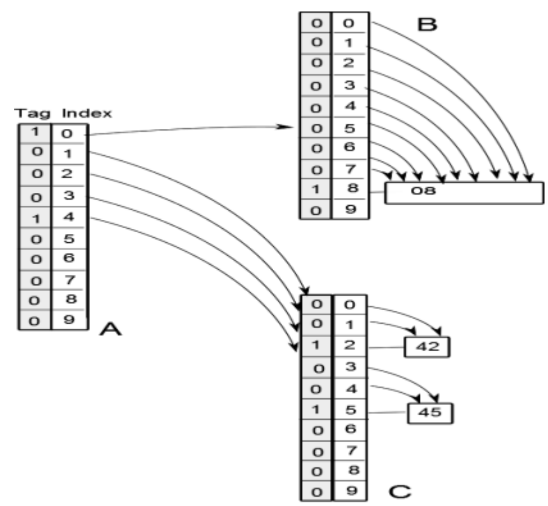

Fig. 2: A sample threaded trie

BITS-trees one at each level, and interlinking the trees of two consecutive levels in a specified manner which are due to the following definitions. These multi-level BITS-trees are termed here as BITS-kd trees.

Definition 3: Given a point p = $(x_1, x_2, \ldots, x_k)$ and an integer $l \leq k$, the head of p and tail of p are defined, respectively as head(p,l) = $(x_1, x_2, \ldots, x_k)$ and tail(p,l) = $(x_{k-l+1}, x_{k-l+2}, \ldots, x_k)$. Also, having (head(p,l), $y_1, y_2, \ldots, y_m$) = $(x_1, x_2, \ldots, x_l, y_1, y_2, \ldots, y_m)$, leads to (head(p,l), tail (p, k-l)) = p.

Definition 4. Given S as the set of points in R^k and |S| = n, the set S_1 is defined as $S_1 = \bigcup_{i=1}^{n}\{(x) \mid p \in S \text{ and } head(p,1) = (x)\}$. That is, S_1 is the set of distinct x values of the points in S. In general, $S_j = \bigcup_{i=1}^{n}\{(x_1, x_2, \ldots x_j) \mid p \in S$ and head(p,j) = $(x_1, x_2, \ldots, x_j)$ where $1 \leq j \leq k$.

Definition 5: For a point p = $(x_1, x_2, \ldots, x_j)$ in S_j, the term x_j is said to be the dimensional value of p as the set of points in S_j is used to construct jth level BITS-tree.

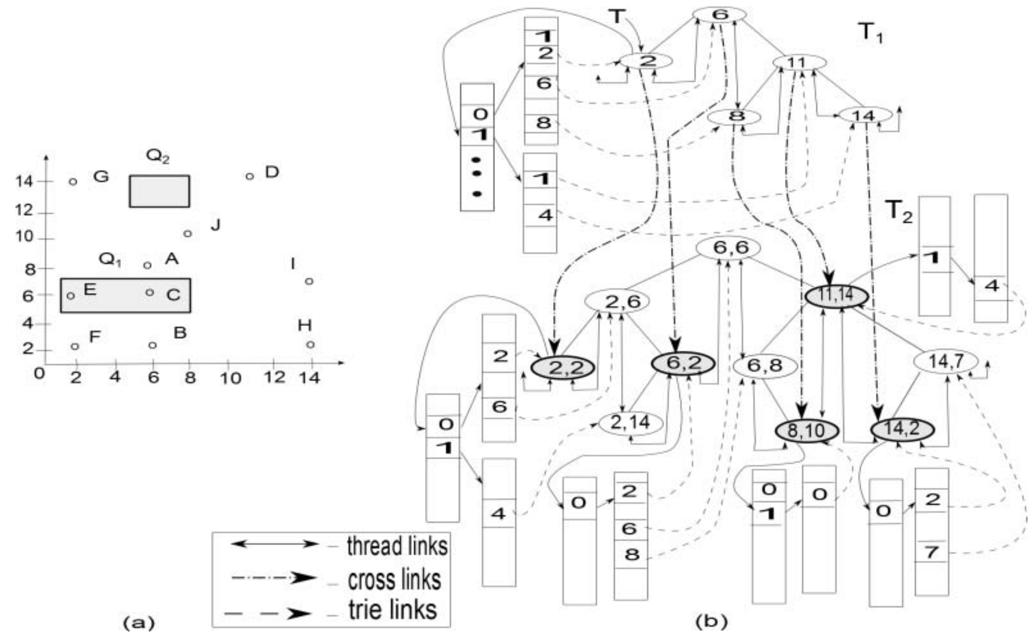

Fig. 3: A BITS 2d-tree: a) Spatial representation of points; b) BITS-2d tree for points shown in (a)

Definition 6: A BITS kd-tree is a multi-level tree which is constructed as follows:

- Create separate BITS- trees, T_j for each S_j, $1 \leq j \leq k$
- Let $X_j = (x_1, x_2, ..., x_j)$ Now, for each node, say $v_j = (X_j, L, -)$ of $T_j, 1 \leq j \leq k$, the list L points to the node $v_{j+1} = (X_j, x'_{j+1}), -, -)$ of T_{j+1} where $x'_{j+1} = \min\{x_{j+1} \mid \text{head}(p, j) = X_j$ and $(\text{head}(p,j), x_{j+1}) \in S_{j+1}\}$. We term these links as cross links and the node v_{j+1} as a cross link node in T_{j+1}
- In T_1, there is only one cross link node which is the first node in the in order sequence and the tree pointer always points to this node
- For each node $v = ((X_{j-1}, x'_j), -, T)$ in T_j, $1 \leq j \leq k$, T is pointer to the threaded tree if v is a cross link node and otherwise T is set to be null
- For every cross link node $v_j = ((X_{j-1}, x'_j), -, T)$ in T_j, the data node of T for a key, say k', points to the node $((X_{j-1}, k'), -, -)$ in T_j. That is, T provides links to the nodes in $\{((X_{j-1}, x_j), -, -) \mid (X_{j-1}, x_j) \in S_j\}$ and these links are termed as trie links.

A BITS 2d-tree for the sample points in Fig. 3a is shown in Fig. 3b. Since, BITS kd-trees are multi-level trees with binary inorder threaded search trees at each level, the height of the trees at each level is O(log n). Also, each node in T_{i-1} has a cross link to a node in T_j which has the least value for the ith co-ordinate with respect to the head value of the node in T_{i-1}. Note here that at least one such point exists. This link is useful to locate a list of collinear points in the ith dimension, associated with a point in T_{i-1}. Also, the trie links are useful to locate a point in a given range window in constant time. The cross link and trie links also make the structure much suitable to address range queries efficiently.

Normally, kd-trees perform insertion by a simple comparison between the respective co-ordinates at each level. However, deletion is tedious due to candidate replacement. This is because, candidate for replacement can be anywhere in the subtree. Also, it requires a little more work when the right subtree is empty. Now, to find a candidate for replacement, it is required to find the smallest element from the left subtree to avoid violation of the basic rules of kd-trees and then it is required to perform a swap of left and right subtrees as many possible candidate keys exist in the left subtree. To handle such a situation, we make use of a collection of BITS-trees, one for each dimension. Here, deleting a point may or may not require a replacement but if so, it is only the inorder successor and that can be located in $\theta(1)$ time as inorder links exist for each node. Also, the cross links that exist between two consecutive levels, practically provide a faster search on next level trees. Another advantage of this structure is that when a node is pruned out at a particular level, it need not be considered in the subsequent levels. That is, nodes that have head values as these will be ignored in the subsequent levels. To the best of our knowledge, there is no such structure using multi-levels of balanced binary search trees with two-way threads introduced in this work, for storing point data and to perform range search efficiently.

Range search for window query: Given a rectangular range in the form of a window, a range query finds all points lying within this window. Let $[x_1: x_2]\times[y_1: y_2]$ be a given query range. First, we use a trie stored in the first node, which is the only cross link node, in the tree at level l to find the smallest point larger than or equal to x_1 in S_1. The non-existence of such a point is determined from the trie itself. On the other hand, once such a point p is located, subsequent points that fall within $[x_1: x_2]$ can be determined using the inorder threads as the inorder sequence is in sorted order. Let us say that the reported set of points as S'. However, if the dimensional value of the point T_2 is greater than x_2, it implies absence of required candidates.

Now, using cross links of the node in T_1 that corresponds to each point in S', further search is performed at T_2 in a similar fashion. Note that each cross link node in T_2 has a trie structure that supports quick access to a node in T_2 where the dimensional value is in $[y_1: y_2]$. In case, the dimensional value of the cross link node is within $[y_1: y_2]$, the respective trie structure need not be looked into, instead the inorder threads are used to find the remaining candidates.

Example 1: For instance, let us consider Fig. 3 with search range $[1{:}8]\times[5{:}7]$. First, we use the only cross link node present in T_1. As its dimension value, i.e., 2 lies within the range [1:8], we do not use the respective trie. Instead, we use the inorder threads to identify the candidate points, which are 2, 6 and 8. Now, for each of these candidates, further search is continued respectively from (2,2), (6,2) and (8,10) in T_2 as these are the corresponding cross link nodes. Now, by looking at the tries of these cross link nodes we find a point whose dimensional value is the smallest one is [5:7]. Thus, tries of (2,2) yields (2,6), (6,2) yields (6,6) and (8,10) yields nothing. Further, by performing inorder traversal from (2,6) and (6,6), the final reported points for Q_1 are E(2,6) and C(6,6). Also, for Q_2 i.e, $([5{:}8]\times[12{:}14])$, no points will be reported.

Notice that one can stop the search at T_1 without traversing T_1 if there is no candidate node in T_1 within the given range. This is also applicable in k-d trees because if there is no candidate node in the higher tree, the lower level trees need not be searched. Thus, this structure prunes the search in some cases and thereby practically reduces the time for reporting a query.

A range search on k-dimensional points can be performed by extending the search on T_3, T_4, ..., T_k similar to that of T_2 as in the case of 2d range search. However, in T_1 and T_2, we need to perform the search as described for 2d range search. That is, when we take the query range as $[x_1:x_1']\times[x_2:x_2']\times\ldots[x_k:x_k']$, the search is performed to find candidates within the range of $[x_1:x_1']$ in T_1, $[x_2:x_2']$ in T_2, $[x_3:x_3']$ in T_3 and so on. Finally, the points reported from T_k will be in Q. It is important to note that the search requires comparison of keys within the given range of the particular co-ordinate dimension in each of T_1, T_2, ..., T_k. This simplifies subsequent searches at the next level.

RESULTS AND DISCUSSION

Two dimensions: Given a set of two dimensional points in R^2, a two-level tree (BITS 2d-tree) is constructed in O(n) time as a point may require at most two insertions, one at T_2 and the other at T_2. But, the position at which insertion is to be made in T_1 and T_2 could be determined in constant time as described in the proof of Lemma 5. Thus, to insert n nodes requires O(n) time. Also, it may be required to create a cross link for each node of T_1 in the case of BITS 2d-tree. Since,T_1 cannot have more than n points, the number of cross links created cannot exceed n. Also, the number of trie links created cannot exceed the number of nodes in T_1 and T_2 which is O(n). Moreover, construction of a trie requires only constant time as the height of the trie is constant due to fixed size of the key. Thus, all these factors lie within O(log n) for each insertion. Regarding space requirements in a BITS 2d-tree, it is O(n) as the second tree is the one that contains all the n points and fewer or equal number of points in the first tree. Also, the number of trie nodes is O(n) as the height of a trie is constant which is due to the size of (number of digits) of the key. Thus, we obtain the following lemma.

Lemma 1: Construction of BITS 2d-tree for O(n) points requires O(n) time and O(n) space.

Now, searching a candidate node in T_1 is done through the trie in T_1 and that requires only constant time as the height of the trie is fixed. Once such a point is identified, subsequent points are identified through inorder threads. Thus, for identifying candidate points, it takes only $\theta(t_1)$ time, if there are t_1 candidate points in T_1. Now, using cross links of each of these nodes, we can locate the required tries in constant time and further search is to be done in a similar fashion as described earlier. Thus, it leads to the following lemma.

Lemma 2: Range search for window query using BITS 2d-tree can be addressed in $\theta(t)$ time where t stands for the number of points reported.

Higher dimensions: A straight forward extension of BITS 2d-tree to k-dimensions is made easy by connecting (cross links) to the corresponding nodes in the tree at next

Table 1: Theoretical comparison of kd-trees, divided k-d trees, range trees, k-d Range DSL-trees, layered range trees and the proposed BITS kd-trees

Description	Storage	Construction	Update	Range search
kd-Trees (Bentley, 1975)	O(n)	O(n log n)	$O(\log^k n)$	$O(n \log^{1-1/k}+t)$
Divided k-d trees (van Kreveld and Overmars, 1991)	O(n)	O(n log n)	$O(\log^{k-1} n)$	$O(n^{1-1/k}\log^{1/k} n+t)$
Range trees (Bentley, 1980)	$O(n \log^{k-1} n)$	$O(n \log^{k-1} n)$	$O(\log^k n)$	$O(\log^k n+t)$
kd-Range DSL-Trees (Lamoureux and Nicolson, 1995)	$O(n \log^{k-1} n)$	$O(n \log^k n)$	$O(n \log^{k-1} n)$	$O(\log^k n+t)$
Layered Range Trees (Willard, 1978)	$O(n \log^{k-1} n)$	$O(n \log^{k-1} n)$	$O(\log^k n)$	$O(\log^{k-1} n+t)$
BITS kd-Trees	O(n)	O(n)	Ins. $\theta(1)$ Del. O(log n)	$\theta(t)$

n = No. of points, k-dimensions, t = No. of points reported

level. Unlike, range trees (Bentley, 1980) which build another range tree at a given node from the main tree, we maintain the trees $T_1, T_2, ..., T_k$ dimension-wise such that the inorder traversal provides an ordered sequence of points stored in the tree. This definitely reduces the overall time taken for range search across k-dimensions. As described in the previous section, the time required to find a candidate point in any T_i, $1 \le i \le k$ is only a constant. Thus, it leads to the following lemma.

Lemma 3. Let S be a set of points in k-dimensional space, $k \ge 1$. A range search on BITS kd-tree reports all points that lie within the rectangular query range in $\theta(t)$ time, where t is the number of points reported.

Lemma 4: Given a set of n points, a BITS kd-tree can be constructed in O(n) time and O(n) space.

Proof: Since, we construct $T_1, T_2, ..., T_k$, such that T_k at level k has at most n nodes, it follows that $N(T_i) \le N(T_{i+1})$, $1 \le i \le k$ and $N(T_k) = n$ where $N(T_i)$ is the number of nodes in T_i. Note that levels correspond to dimensions and hence may be used interchangeably. Also, the number of trie nodes is k O(n) as its height is constant. Therefore, for k levels, a BITS kd-tree uses O(n) storage in the worst case as k is a constant. Now, construction of BITS kd-tree is considered as a sequence of insertions. Each insertion, may or may not alter T_1, $1 \le i \le k$, a BITS tree of a particular level. However, if a BITS-tree T_i is altered, due to insertion, all trees T_j will be altered. Let j be the least index such that the tree T_{j+1}, T_{j+2}, T_k is altered. Thus, for $T_1, T_2, ..., T_{j-1}$ with trie links and cross links, one can determine that the required values are already stored in those trees within constant time. Now, from a particular cross link in T_{j-1} followed by a trie link in T_j, one can find a position for the new value in T_j. This requires only constant time. Then, while inserting the value if the tree is unbalanced, atmost one rotation is required to balance the tree. So, for T_j too, it requires constant time. Let n_j be the new node inserted in T_j. Now, by taking cross link of inorder successor of n_j, one can determine the position of the new node in T_{j+1} and that as inorder predecessor of cross link node of inorder successor of n_j. This new node in T_{j+1} need to have a trie, which again be created in constant time. Then, the process is to be continued for $T_{j+2}, ..., T_k$. Here, updation in each T_i, $1 \le i \le k$ takes only constant time and hence each insertion takes $\theta(1)$ time. So, construction of BITS kd-tree for n points requires O(n) time.

Lemma 5: Insertion and deletion of a point in a BITS kd-tree can be respectively done in $\theta(1)$ and O(log n) time.

Proof: As per the description given in the proof of Lemma 4, insertion of a point in BITS kd-tree takes only $\theta(1)$ time. But for deletion, finding a node to be removed from a BITS-tree requires only constant time. However, if that node is not a leaf node a cascading replacement with inorder successor is required until reaching a leaf node to be removed physically. Certainly, the number of such replacements to be done cannot exceed O(log n). After that, it may require a sequence of rotations on the path from the physically removed leaf to the root and that too in at most O(log n) rotations. So, deletion of a point in BITS kd-tree requires O(log n) time.

Performance: Table 1 summarizes the performance of kd-trees, divided k-d trees, range trees, kd-range DSL-trees, layered range trees and the BITS kd- tree proposed in this work. Furthermore, our theoretical comparison of the BITS kd-tree is made with kd-trees adapted for internal memory (pointer machine model) and not with any of the other bulk loading kd-trees(RAM model). The results give an $\theta(t)$ query time using the BITS kd-tree that shows a reduction in time as compared to the existing bounds. Since we try to capitalize on the efficiency of balanced search trees at all the levels by using cross links and trie links, we ensure that the number of nodes visited during a range query is considerably reduced in BITS kd-tree. Observe that the storage is increased from O(n) in kd-trees to $O(n \log^{k-1} n)$ in range trees while BITS kd-tree still maintains an O(n). Notice that the update time for BITS kd-tree has been reduced considerably. To summarize, although the storage requirements of BITS kd-tree are comparable to k-d trees, divided k-d trees, the construction and update time are improved considerably. Moreover, the overall query time is improved to $\theta(t)$ time where t is the number of points reported as it prunes points falling outside the query region for each dimension.

CONCLUSION

A BITS kd-tree for storing k-dimensional points having update and query operations efficiently than kd-trees is proposed. The main advantage of this tree is that it effectively handles the collinear points. As a result, number of nodes visited during search is much less compared to other kd-tree variants that are either not height balanced or update operation is complex. In the case of height balanced kd-trees, having better search efficiency, insertion is tedious. A k-d range DSL tree gives a logarithmic amortized worst case search time with efficient updates mainly for partial match queries and not for window queries. In BITS kd-tree, overall insertion time is $\theta(1)$. Moreover, points can be dynamically updated at each level. Since, co-ordinate dimensions at each level are distributed and using threaded tries, we quickly find points falling within the query range. Also, points falling above and below the search range are pruned efficiently using cross links to the next level and inorder threads similar to the BITS-tree. In addition, threaded tries introduced in this work link the node, having cross link, by means of trie links to find the points within the given range in constant time. Therefore, range search for points in a rectangular region using BITS k-d tree takes $\theta(t)$ time where t is the number of points reported, and therefore the logarithmic factor in earlier worst case bounds is reduced. Hence, it is definitely a remarkable improvement over $O(n^{1-1/k}+t)$ of kd-trees and $O(\log^k n+t)$ time of k-d range DSL trees.

REFERENCES

Afshani, P., L. Arge and K.G. Larsen, 2012. Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model. Proceedings of the 28th Annual ACM Symposium on Computational Geometry, June 17-20, 2012, ACM, New York, USA., ISBN: 978-1-4503-1299-8, pp: 323-332.

Alstrup, S., G.S. Brodal and T. Rauhe, 2000. New data structures for orthogonal range searching. Proceedings of the 41st Annual Symposium on Foundations of Computer Science, November 12-14, 2000, IEEE, Redondo Beach, California, ISBN: 0-7695-0850-2, pp: 198-207.

Bentley, J.L. and J.B. Saxe, 1979. Decomposable searching problems. Inf. Process. Lett., 8: 5-9.

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

Bentley, J.L., 1979. Multidimensional binary search trees in database applications. Software Eng. IEEE. Trans., 4: 333-340.

Bentley, J.L., 1980. Multidimensional divide-and-conquer. Commun. ACM., 23: 214-229.

Chan, T.M., K.G. Larsen and M. Patrascu, 2011. Orthogonal range searching on the RAM, revisited. Proceedings of the 27th Annual Symposium on Computational Geometry, June 13-15, 2011, ACM, New York, USA., ISBN: 978-1-4503-0682-9, pp: 1-10.

Crespo, M.M.P., 2010. Design, analysis and implementation of new variants of Kd-trees. Master Thesis, Polytechnic University of Catalonia, Barcelona, Spain.

De Berg, M., O. Cheong, M. Van Kreveld and M. Overmars, 2008. Computational Geometry: Algorithms and Applications. 3rd Edn., Springer, New York, ISBN-13: 9783540779735, Pages: 386.

Devroye, L., J. Jabbour and Z.C. Cura, 2000. Squarish kd trees. SIAM. J. Comput., 30: 1678-1700.

Easwarakumar, K.S. and T. Hema, 2015. BITS-tree-an efficient data structure for segment storage and query processing. Int. J. Comput. Technol., 11: 3108-3116.

Kreveld, M.J.V. and M.H. Overmars, 1991. Dividedk-d trees. Algorithmica, 6: 840-858.

Lamoureux, M.G. and B.G. Nickerson, 1995. Deterministic skip lists for K-dimensional range search. University of New Brunswick Technical Report TR95-098, Revision, 1.

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

Nekrich, Y., 2009. Orthogonal range searching in linear and almost-linear space. Comput. Geom., 42: 342-351.

Nelson, R.C. and H. Samet, 1986. A consistent hierarchical representation for vector data. Proceedings of the Conference on ACM SIGGRAPH Computer Graphics, August 4, 1986, ACM, New York, USA., ISBN:0-89791-196-2, pp: 197-206.

Nilsson, S. and M. Tikkanen, 2002. An experimental study of compression methods for dynamic tries. Algorithmica, 33: 19-33.

Orenstein, J.A., 1982. Multidimensional tries used for associative searching. Inf. Process. Lett., 14: 150-157.

Preparata, F. and M.I. Shamos, 1985. Computational Geometry: An Introduction. Springer-Verlag, New York.

Samet, H., 1974. Fundamentals of Multi-Dimensional and Metric Data Structures. Academic Press, New York, USA..

Samet, H., 1990. The Design and Analysis of Spatial Data Structures. Addition Wesley, New York.

Tropf, H. and H. Herzog, 1981. Multidimensional range search in dynamically balanced trees. Angew. Inf., 2: 71-77.

Willard, D., 1978. New Data Structures for Orthogonal Queries. Harvard University, Harvard University, Pages: 38.