Colored Range Searching on Internal Memory

Haritha Bellam[1], Saladi Rahul[2], and Krishnan Rajan[1]

[1] Lab for Spatial Informatics, IIIT-Hyderabad, Hyderabad, India
[2] Univerity of Minnesota, Minneapolis, MN, USA

Abstract. Recent advances in various application fields, like GIS, finance and others, has lead to a large increase in both the volume and the characteristics of the data being collected. Hence, general range queries on these datasets are not sufficient enough to obtain good insights and useful information from the data. This leads to the need for more sophisticated queries and hence novel data structures and algorithms such as the *orthogonal colored range searching (OCRS)* problem which is a *generalized* version of orthogonal range searching. In this work, an efficient main-memory algorithm has been proposed to solve $OCRS$ by augmenting k-d tree with additional information. The performance of the proposed algorithm has been evaluated through extensive experiments and comparison with two base-line algorithms is presented. The data structure takes up linear or near-linear space of $O(n \log \alpha)$, where α is the number of colors in the dataset ($\alpha \leq n$). The query response time varies minimally irrespective of the number of colors and the query box size.

1 Introduction

Multi-attribute queries are becoming possible with both the increase in kind of attributes the datasets are able to store and also the processing power needed to do so. In addition to specific queries across the multiple attributes, there is an increasing need for range queries across these attributes especially in fields like GIS, business intelligence, social and natural sciences. In most of these one needs to identify a set of classes that have attribute values lying within the specified ranges in more than one of these attributes. For instance, consider a database of mutual funds which stores for each fund its annual total return and its beta (a real number measuring the fund's volatility) and thus can be represented as a point in two dimensions. Moreover, the funds are categorized into groups according to the fund family they belong to. A typical two-dimensional orthogonal colored range query is to determine the families that offer funds whose total return is between, say 15% and 20%, and whose beta is between, say, 0.9 and 1.1 [8]. An another example can be to identify potentially suitable locations for afforestation programs and the choice of right vegetation types for these locations. The spatial database can contain multiple themes like soil class, weather related parameters (temperature, relative humidity, amount of precipitation), topography and ground slope, while for each vegetation type there exists a suitable range of values in these themes/attributes. In such cases, one is interested

S.-g. Lee et al. (Eds.): DASFAA 2012, Part II, LNCS 7239, pp. 111–125, 2012.

in finding the best match between the two, leading to a need to answer a set of colored range queries. Generally speaking, in many applications, the set S of input points come aggregated in groups and we are more interested in the groups which lie inside q rather than the actual points. (a group lies inside q if and only if at least one point of that group lies inside q.) For convenience, we assign a distinct color to each group and imagine that all the points in the group have that color. This problem is known as *colored range searching*. In this paper, we specifically consider the problem of *orthogonal colored range searching* (see Figure 1) where the query is an orthogonal box in d-dimensional space (i.e $q = \Pi_{i=1}^{d}[a_i, b_i]$) and propose a algorithm to effectively deal with it.

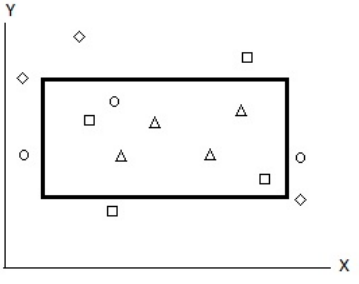


Fig. 1. A set of ten colored points lying in a plane. For the given orthogonal query rectangle, a standard orthogonal range query would report all the points lying inside the query rectangle. However, an orthogonal colored range searching (OCRS) query would report the unique colors of the points lying inside the rectangle.In this figure, each color is represented with a different symbol.The output for this query would be △, □ and ○. Note that ◇ is not reported.

Orthogonal colored range searching ($OCRS$) can be described in an SQL-like language (basically a GROUP-BY operation). However, the current DBMS's do not provide with efficient query processing techniques for $OCRS$. $OCRS$ query will be executed by first executing the standard orthogonal range searching query. This will report all the points lying inside q. Then all the reported points are scanned to filter out the unique colors. In many real-life applications, there could be a huge difference between the number of points lying within q and the number of unique colors among them. This can lead to poor query time.

$OCRS$ and the general class of colored intersection problems [8] have been extensively studied in the field of Computational Geometry. Efficient theoretical solutions for $OCRS$ though have been provided on internal memory (or main memory) with the objective of providing worst-case optimal performance. However they suffer from the following shortcomings:

1. Most of these solutions are good in theory but are extremely difficult to implement.
2. The space of the data structures proposed increase exponentially in terms of the dimension size. Building linear or near-linear size data structures is a requirement in In-memory database systems (IMDS) that are steadily growing

[1]. In fact we implemented one of these theoretical solutions [7] but had to be discarded since it was not fitting into the main-memory for the datasets we tested on (see Figure 3).
3. For most of the practical applications we need not come up with solutions which need to be optimal even in the worst case scenario. In practice the worst case scenario occurs *extremely rarely*.

We seek to to build a solution which works well for *most of the cases/scenarios*. The problem that will be addressed in this paper is formally stated as-

– Preprocess a set S of n colored points lying in d-dimensional space (or $\mathbb{R}^d$) into an index structure, so that for any given orthogonal query box $q{=}\Pi_{i=1}^{d}[a_i, b_i] \subseteq \mathbb{R}^d$, all the distinct colors of the points of S lying inside q need to be reported efficiently.

Since we are dealing with main-memory, the main focus in this paper is on building space-efficient data structures for answering orthogonal colored range searching query. The following are the main contributions of the paper:

– We come up with non-trivial techniques for building the data structure and for answering the query algorithm of orthogonal colored range searching ($OCRS$) on main-memory. The objective is to *use minimal space* while maintaining efficient query time.
– Detailed experiments and theoretical analysis have been carried out to show the performance of our technique against two base-line algorithms which can be used to solve the orthogonal range searching problem in relation to the distribution that the data exhibits, the query size and the number of colors in the data. These two base-line algorithms are described later in the paper.

2 Related Work

Orthogonal range searching is one of the most popular and well studied problem in the field of computational geometry and databases. The formal definition is the following: "Preprocess a set S of n points lying in d-dimensional space (or $\mathbb{R}^d$) into an index structure, so that for any given orthogonal query box $q{=}\Pi_{i=1}^{d}[a_i, b_i] \subseteq \mathbb{R}^d$, all the points of S lying inside q need to be reported/counted efficiently". There have been a numerous data structures proposed in the computational geometry literature to handle orthogonal range searching query. Traditionally, the researchers in computational geometry have been interested in building solutions for this problem which aim at coming up worst-case efficient query time solutions, i.e., ensuring that the query time is low for *all* possible values of the query. The most popular among them are the range-trees [6] and the k-d tree [6,5]. The original range-tree when built on n points in d-dimensional space took $O(n \log^d n)$ space and answered queries in $O(\log^{d-1} n + k)$ query time, where k are the number of points lying inside the query region q. By using the technique of fractional cascading we can reduce the query time by a log factor [6].

In a dynamic setting, the range-tree uses $O(n \log^{d-1} n)$ space, answers queries in $O(\log^{d-1} n \log\log n + k)$ time and handles updates in $O(\log^{d-1} n \log\log n)$ time. A k-d tree when built on n points in d-dimensional space takes up *linear space* and answers a range-query in $O(n^{1-1/d} + k)$ time. Updates can be handled efficiently in it. A detailed description of k-d tree's construction and query algorithm shall be provided later.

In the field of databases there have been a significant number of index structures proposed to answer an orthogonal range-query. These are practical solutions which are optimized to work well for the average case queries (with the assumption that the worst-case query will occur rarely). Perhaps no other structure has been more popular than the *R-tree* proposed in 1984 by Guttman [9]. It's main advantage arises from the fact that it is versatile and can handle various kinds of queries efficiently (range-queries being one of them). K-d tree also enjoys similar benifits of being able to handle different kinds of queries efficiently. Varaints of R-tree such as R* tree [4], R+ tree [13], Hilbert tree and Priority R-tree are also quite popular index structures for storing multi-dimensional data.

Orthogonal colored range searching (OCRS) happens to be a *generalization* of the orthogonal range searching problem. Janardan et al. [10] introduced the problem of OCRS. Gupta et al. [7] came up with dynamic (both insertions and deletions) solutions to this problem in 2-dimensional space and a static solution in 3-dimensional space. The best theoretical solution to this problem in d=2 has been provided by Shi et al. [14] which is a static data structure taking up $O(n \log n)$ space and $O(\log n + k)$ time, where 'k' is the number of colors lying inside the query rectangle. For d=3, the only known theoretical solution takes up $O(n \log^4 n)$ space and answers query in $O(\log^2 n + k)$ time [7]. For $d > 3$, there exists a data structure which answers queries in $O(\log n + k)$ time but takes up $O(n^{1+\epsilon})$ space which from a theoretical point of view is not considered optimal ($O(n$ polylog $n)$ is desirable). As stated before, the objective in the computational geometry field has been to come up with main-memory algorithms which are optimal in worst-case scenario. A good survey paper on OCRS and other related colored geometric problems is [8].

Agarwal et al. [3] solved $OCRS$ on a 2-dimensional grid. Recently, there has been some work done on range-aggregate queries on colored geometric objects. In this class of problems, the set S of geometric objects are colored and possibly weighted. Given a query region q, for each distinct color c of the objects in $S \cap q$, the tuple $\langle\, c, \mathcal{F}(c) \,\rangle$ is reported where $\mathcal{F}(c)$ is some function of the objects of color c lying inside q [11,12]. Examples of $\mathcal{F}(c)$ include sum of weights, bounding box, point with maximum weight etc. In [11,12], theoretical main-memory solutions have been provided.

3 Existing Techniques to Solve OCRS

In the database community OCRS problem is solved by using the *filter and prune* technique. Two base-line algorithms are described which follow this paradigm. Then we shall discuss about the theoretical solutions which emerged from computational geometry for answering OCRS. Based on these discussions we shall

motivate the need for a practical data structure and algorithm which will work efficiently in a main-memory or internal memory environment.

3.1 Base-line Algorithms

Here we shall describe two base-line algorithms with which we shall compare our proposed method. These two algorithms solve the standard orthogonal range searching problem. We will describe how they can be modified to answer the orthogonal *colored* range searching problem.

1. **Plain Range Searching (PRS)** In this method, we build a k-d tree on all the points of S. Given an orthogonal query box q, we query the k-d tree and report all the points of S lying inside q. Then we filter out the unique colors among points that are lying inside q. The space occupied by a k-d tree is $O(n)$. The query time will be $O(n^{1-1/d} + |S \cap q|)$, where $S \cap q$ is the set of points of S lying inside q. If the ratio of $|S \cap q|$ and the number of unique colors is very high, then this technique will not be efficient.
2. **Least Point Dimension (LPD)** A set S of n points lie in a d-dimensional space with some color associated to each point. For each dimension i we project S onto dimension i, then we sort the points of S w.r.t coordinate values in dimension i and store them in an array A_i. Given an orthogonal query $q = \Pi_{i=1}^{d}[a_i, b_i]$, we decompose the query q into intervals corresponding to each dimension (interval corresponding to dimension i will be $[a_i, b_i]$). Then we do a binary search on all the arrays A_i ($\forall\ 1 \leq i \leq d$) with $[a_i, b_i]$ to find the number of points of S lying inside it. The array A_l, in which the total number of points is least is chosen. Each point of A_l which occurs within $[a_l, b_l]$ is also tested for its presence inside q. Among all the points which pass the test, the unique colors of these points are filtered out. Query time for LPD is $O(d \log n + \beta)$ where β is the number of points in A_l which lie inside $[a_l, b_l]$. Note that the performance of LPD algorithm is dependent on the value of β. If the ratio of β and the number of unique colors is high, then the performance of LPD will not be good.

3.2 Existing Theoretical Solutions

As mentioned before in Section 2 (related work), there have been theoretical solutions proposed to answer OCRS. Only a few of these solutions can be implemented and tested. Most of them are meant for theoretical interest and are impossible to implement. We implemented the solution proposed by Gupta et al. [7] for $d = 2$ (semi dynamic solution which handles only insertions). Theoretically it takes up $O(n \log^2 n)$ space, $O(\log^2 n + k)$ query time (k is the number of unique colors reported) and $O(\log^3 n)$ amortized insertion time. We observe that for large datasets $O(n \log^2 n)$ space is not acceptable as that would mean storing $O(\log^2 n)$ copies of each data item in the internal memory. Experiments on real-life and synthetic datasets confirmed our intuition that this data structure takes up a lot more space than the base-line algorithms and the proposed

solution in this paper (see figure 3). Therefore, *this solution was discarded*. For higher dimnesions ($d > 2$), the theoretical solutions get too complicated to be implemented and tested.

4 Proposed Algorithm

As we are trying to build main-memory structures, the highest priority is to minimize the space occupied while trying to keep the query time competitive.

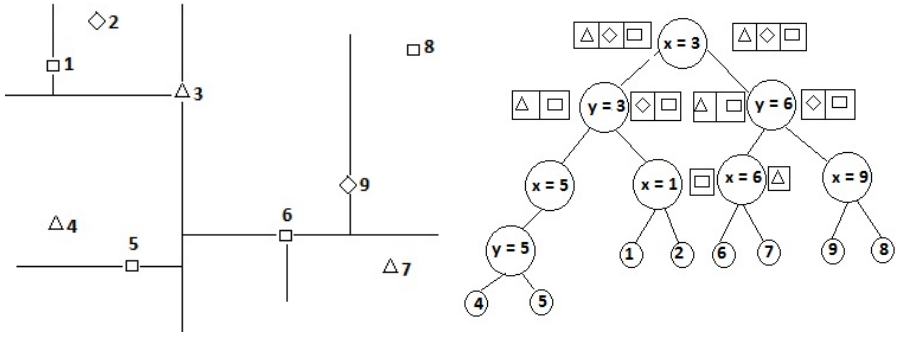
4.1 Data Structure

In this section we construct a data structure named BB k-d tree, which is based on a regular k-d tree but is augmented with additional information at each internal node. We chose k-d tree as our primary structure since it takes up linear space for indexing points lying in any dimension and though it has a poor theoretical query performance, it does well in practice [15,5].

We shall first describe the structure for a 2-dimensional scenario The primary structure of BB k-d tree is a conventional k-d tree. A splitting line $l(v)$, here choosen as a median, is stored at every node (v) which partitions the plane into two half-planes. We denote the region corresponding to a node v by $region(v)$. The region corresponding to the left child and right child of the node v are denoted as $region(lc(v))$ and $region(rc(v))$, where $lc(v)$ and $rc(v)$ denote the left and right children of v, respectively.

At each internal node v, apart from the regular information two height-balanced binary search trees $\mathcal{T}_l$ and $\mathcal{T}_r$ are stored. $\mathcal{T}_l$ is built as follows: Let c be one of the colors among the distinct colors of the points lying in the subtree rooted at $lc(v)$. The bounding box of the points of color c in the subtree of $lc(v)$ is calculated and kept at a leaf in $\mathcal{T}_l$. A bounding box for a point set in the plane is the rectangle with the smallest measure within which all the points lie. This process is repeated for each color c which lies in the subtree of $lc(v)$. The colors stored in the leaves of $\mathcal{T}_l$ are sorted lexicographically from left to right. To make navigation easier, each leaf in $\mathcal{T}_l$ is connected to its adjacent leaves. This forms a doubly linked list among the leaf nodes of $\mathcal{T}_l$. Also, the root node of $\mathcal{T}_l$ maintains a special pointer pointing to the leftmost leaf of $\mathcal{T}_l$. Similarly, $\mathcal{T}_r$ is constructed based on the points stored in the subtree of $rc(v)$.

BB k-d tree can also be built for point sets in 3 or higher-dimensional space. The construction algorithm is very similar to the planar case: At the root, we split the set of points into two subsets of roughly the same size by a hyperplane perpendicular to the x-axis. In other words, at the root the point set is partitioned based on the first coordinate of the points. At the children of the root the partition is based on the second coordinate, at nodes of depth two on the third coordinate, and so on, until depth $d - 1$ where we partition on the last coordinate. At depth d we start all over again, partitioning on first coordinate. The recursion stops when there is only one point left, which is then stored at a leaf. The binary search trees ($\mathcal{T}_l$ and $\mathcal{T}_r$) at each internal node are constructed in the same manner as described above.



(a) Set of 9 points lying on a plane

(b) BB-tree corresponding to these nine points. Note that all the arrays are not represented for the sake of clarity

Fig. 2. An example of a BB-tree. At every node there are two arrays. The left/right array contains the unique colors present in the left/right subtree along with their respective bounding boxes.

In the example shown in Fig 2 there are 9 points and a color has been assigned to each point. The construction of the K-d tree on these points has been showed in Fig 2(a). At the root we split the point set P with a vertical line $x = 3$ into two subsets of roughly equal size. At the left and right children we split the point set using horizontal median lines. In this way we build a traditional k-d tree first. At each internal node the bounding box of all the colors lying in the left (resp. right) subtree are stored in $\mathcal{T}_l$ (resp. $\mathcal{T}_r$). For example, the left child of the root node has the bounding box of $\triangle$ points 3 and 4, bounding box of $\square$ points 1 and 5, bounding box of the $\diamond$ point 2, are computed and stored at the left binary search tree ($\mathcal{T}_l$) of the of root node of k-d tree. Similarly $\mathcal{T}_r$ is also constructed.

Theorem 1. *A BB-tree with n points and α colors takes $O(n\log\alpha)$ storage*

Proof. To store a set of n points in d-dimensional space, a normal k-d tree uses $O(n)$ space. The space occupied by the internal binary search trees $\mathcal{T}_l$ and $\mathcal{T}_r$ dominates the overall space complexity. Let the number of distinct colors of the points in S be α. We want to find the space occupied in the worst case. The height of a k-d tree will be $O(\log n)$. The height of a node is the length of the longest downward path to a leaf from that node. Consider a node v at height h. If $h \leq \lfloor \log\alpha \rfloor$ (i.e. $2^h \leq \alpha$), then the number of leaves in the subtree rooted at v will be $\leq 2^h \leq \alpha$. Then the size of $\mathcal{T}_l$ and $\mathcal{T}_r$ will be bounded by 2^h (worst case being each point having a unique color). The total space occupied by all the nodes at a particular level h ($\leq \lfloor \log\alpha \rfloor$) will be $O((2^{\log n-h}) \times 2^h) \equiv O(n)$, where $O(2^{\log n-h})$ is the number of nodes at level h. Then the total space occupied by all the internal binary search trees stored at primary nodes having height $\leq \lfloor \log\alpha \rfloor$ will be $\Sigma_{h=0}^{\lfloor \log\alpha \rfloor} O(n) \equiv O(n\log\alpha)$.

Now consider a node v at height $h > \lfloor \log \alpha \rfloor$ (i.e. $2^h > \alpha$). The number of leaves in the subtree at v is bounded by 2^h. However in this case the size of $\mathcal{T}_l$ and $\mathcal{T}_r$ is bounded by $O(\alpha)$. If each node at level h has array size $O(\alpha)$, then the total size of all the arrays at level l will be $O(2^{\log n - h} \times \alpha)$. $\equiv O(\frac{n\alpha}{2^h})$. The overall size of the internal binary search trees added over all levels $h > \lfloor \log \alpha \rfloor$ will be

$$O(\Sigma_{h=\lfloor \log \alpha \rfloor}^{\log n} \frac{n\alpha}{2^h}) \equiv O(n\alpha \Sigma_{h=\lfloor \log \alpha \rfloor}^{\log n} \frac{1}{2^h})$$

$$\equiv O(n\alpha(\Sigma_{h=0}^{\log n} \frac{1}{2^h} - \Sigma_{h=0}^{\lfloor \log \alpha \rfloor - 1} \frac{1}{2^h})) \equiv O(n\alpha(\frac{1}{\alpha} - \frac{1}{n})) \equiv O(n)$$

Therefore, the total size of the BB k-d tree will be $O(n \log \alpha) + O(n) \equiv O(n \log \alpha)$. □

4.2 Query Algorithm

We now turn to the query algorithm. We maintain a global boolean array *outputSet* of size α, which contains an entry for each distinct color of set S. Given a query box q, an entry of *outputSet*[c] being *true* denotes that color c has a point in the query box and we need not search for that color any more inside our data structure. The query algorithm is described in the form of a pseudo-code in Algorithm 1. The input parameters are the current node being visited (v), the orthogonal query box q and a boolean array A. A has a size of α and $A[c]$, $1 \leq c \leq \alpha$, being *true* denotes that we need to search if color c has a point in the subtree of currently visited node (i.e. v), else we need not.

The processing of the query box q commences from the root of the BB k-d tree. Initially, all elements of A are set to *true* and all elements in *outputSet* are set to $false$. If the current node v is not a leaf, then we intialize two arrays A_l and A_r of size α to $false$ (lines $4-5$). A_l (and A_r) are updated later in the procedure and will be used when the children of the current node will be visited.

If $region(lc(v))$ is fully contained in q, then all the colors stored in the leaves of secondary tree $\mathcal{T}_l$ are set to *true* in *outputSet* (lines 6–8). However, if $region(lc(v))$ partially intersects q, then we do the following: Using the special pointer at the root of $\mathcal{T}_l$, we will go to the leftmost leaf of $\mathcal{T}_l$. Then we will check if q contains any bounding box b corresponding to each leaf of $\mathcal{T}_l$ (line 11). The adjacent pointers of each leaf help in navigating through the list of leaves. If a bounding box b (of color c) is fully contained in q then there exists at least one point of color c in q. In this case we will update the *outputSet*[c] to *true* and we need not search for this color anymore (lines 11–12). If a bounding box b partially intersects q, then we need to search for that color c in the subtree of v. So, we will update A_l to *true* (lines 13-14).

The last case is when the query box q and bounding box b of all the points in the subtree of v do not intersect at all. In this case, we need not search for any color in the subtree of v. This is automatically reflected in the arrays A_l (or A_r)

Algorithm 1. SearchBBTree(v, q, A)

Input : A node in BB k-d tree (v) , Query box (q), an array of colors which need to be searched (A)
Output: An array 'OutputSet' which contains the colors lying inside q

```
1  if v is a leaf and point p stored at v lies in q then
2      outputSet[p.color]= true
3  else
4      forall colors i from 1 → α do
5          A_l[i] = false ; A_r[i] = false
6      if region(lc(v)) is fully contained in q then
7          forall colors c in leaves of T_l do
8              outputSet[c]= true
9      else if region(lc(v)) partially intersects q then
10         foreach bounding box b of color c in leaves of T_l where A[c] = true do
11             if q contains b then
12                 outputSet[c] = true
13             else if q partially intersects b and outputSet[c] = false then
14             A_l[c] = true
15     if region(rc(v)) is fully contained in q then
16         forall colors c in leaves of T_r do
17             outputSet[c] = true
18     else if region(rc(v)) partially intersects q then
19         foreach bounding box b of color c in leaves of T_r where A[c] = true do
20             if q contains b then
21                 outputSet[c] = true
22             else if q partially intersects b and outputSet[c] = false then
23             A_r[c] = true
24     if any A_l[c] = true then
25         SearchBBTree (lc(v), q, A_l)
26     if any A_r[c] = true then
27         SearchBBTree (rc(v), q, A_r)
```

as they are intialized to $false$ in the beginning itself. Similar steps are applied for right subtree (lines 15 – 23). If an entry in A_l (or A_r) is $true$, then there is a possibility of existence of that color in the left (or right) subtree of v. So, we shall make a recursive call by passing $lc(v)$ and A_l (lines 24–25). Similarly, if required a recursive call is made by passing $rc(v)$ and A_r (lines 26–27).

4.3 Handling Updates

Insertion and deletion of points can be efficiently handled in the proposed structure. When a point p (having color c) is inserted into BB k-d tree, then an appropriate leaf node is created by the insertion routine of k-d tree [5]. Then we will update all the secondary structures existing on the path (say Π) from the newly created leaf node to the root, in the following manner: At each node $v \in \Pi$, we search for color c in the secondary structures ($\mathcal{T}_l$ and $\mathcal{T}_r$). If no entry of color c exists, then an appropriate leaf is created (in $\mathcal{T}_l$ and $\mathcal{T}_r$) and the bounding box of color c will be the point p. The adjacency pointers are also set appropriately. If the new node happens to be the leftmost leaf (of $\mathcal{T}_l$ or $\mathcal{T}_r$), then the special pointer from the root (of $\mathcal{T}_l$ or $\mathcal{T}_r$) is set to the newly created node. If an entry of color c already exists, then the bounding box of color c is updated. To delete a point p (having color c) from BB k-d tree, we first delete the appropriate leaf node in the primary structure by using the deletion routine of a k-d tree [5]. Before that, at each node v on the path from the leaf node of p to the root we do the following: Search for the color c (in $\mathcal{T}_l$ and $\mathcal{T}_r$). Then update the bounding box of color c. If the bounding box of color c becomes $null$, then that leaf node is removed from ($\mathcal{T}_l$ or $\mathcal{T}_r$). The adjacency pointers are also appropriately adjusted. If the leaf node being removed was the leftmost entry, then the special pointer from the root of ($\mathcal{T}_l$ or $\mathcal{T}_r$) is set to the new leftmost leaf. Next we shall summarize the update time in our structure in a lemma. In the lemma, by random we mean that for each point the coordinate value in each dimension is an independently generated random real number.

Lemma 1. *(Using [5]) The average time taken to insert a random point into the BB k-d tree is* $O(\log^2 n)$. *The average time taken to delete a random point from a BB k-d tree is* $O(\log^2 n)$. *In the worst case, the time taken to delete a point from BB k-d tree is* $O(n^{1-1/d} \log n)$.

5 Experimental Setup

All techniques were implemented in C using the same components. The system used is a 1.66 GHz Intel core duo Linux machine with 4 GB RAM. Programs were compiled using cc. We used both synthetic and real life data sets. The following datasets (both real and synthetic) have been used for our experiments (n will denote the number of data points and d denotes the dimensionality):

a) *Uniform Synthetic Dataset (D1).* n=1,000,000 and d=2.

b) *Gaussian Synthetic Dataset (D2).* n=100,000, d=2 and σ as 0.2% of the number of points.

c) *Gaussian Skewed Synthetic Dataset (D3).* n=100,000 and d=2. The x-coordinate of these points have been assigned using a gaussian function with σ as 10% of the total points and the y-coordinates are produced using a gaussian function with σ 1% of the total points. This helped us in generating a skewed dataset.

d) *Forest Cover real dataset (R1).* The Cover data set contains 581,012 points and is used for predicting forest cover types from cartographic variables. We used the spatial information to get the 2-d points corresponding to the locations. We used the soil type information, which contains 40 distinct category values to assign colors to the points.

e) *U.S Census real dataset (R2).* The US Census Bureau data contains 199,523 records, from which we derived two separate sets: (i) the Census3d, having as dimensions the age, income and weeks worked; and (ii) the Census4d, having as additional dimension dividends from stocks. As categorical attribute we selected, in both cases, the occupation type,that has 47 distinct values in total. We went to [2] to obtain the datasets.

6 Results and Performance Evaluation

In this section we shall look at different kinds of factors which effect the query output time. This section describes the effect of each factor on the performance of the three techniques. In all the datasets, queries are generated using each data point as the center of the query box. The output time per query is obtained by averaging all the query times. We also look at the space occupied by these techniques.

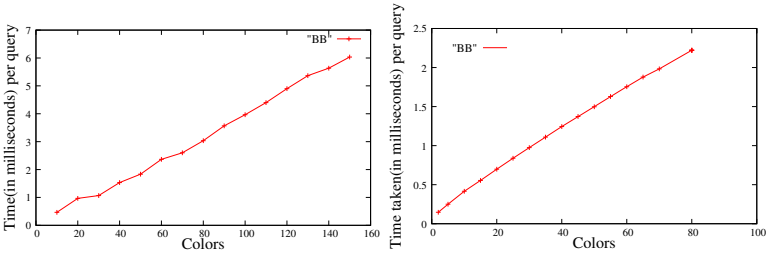
	Forest Cover (in MB)	US Census (3d) (in MB)	US Census (4d) (in MB)
PRS	96	38	45
LPD	152	100	160
BB	510	211	249
Gupta et al.[7]	> 4 GB	> 4 GB	> 4 GB

Fig. 3. Comparision of space occipied by various techniques. We implemented the semi-dynamic solution of Gupta et al.[7] for d=2. For real-life datasets, the size of this data structure exceeded our main memory capacity.

6.1 Comparision of Space Occupied

Theoretically, both PRS and LPD techniques take up $O(n)$ space. However, notice that in LPD we project all the points to each of the d dimensions. Therefore, it is expected to occupy slightly higher space than the PRS technique. This was also observed while testing them on real-life datasets as shown in Figure 3. BB k-d tree occupies $O(n \log \alpha)$ space in any dimensional space. The secondary

information stored at each internal node of a k-d tree leads to a *slight* blow up in the space occupied. In contrast, the data structure proposed by Gupta et al. [7] for d=2 (semi-dynamic solution) could not be loaded to main memory. This was expected as the space complexity of the data structure is $O(n \log^2 n)$. Hence, we could not compute the query time of it and hence discarded this solution.



(a) Uniform dataset D1 (b) Gaussian dataset D2

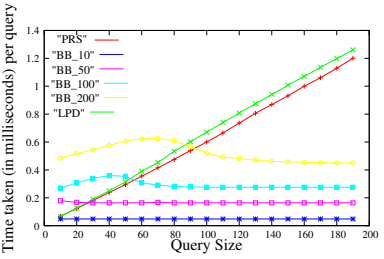
Fig. 4. Performance of BB k-d tree w.r.t the number of colors

6.2 Number of Colors (α)

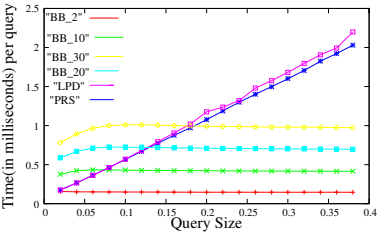
Query response time of PRS and LPD techniques are independent of the number of colors in the dataset. In a BB k-d tree, as the number of colors start increasing in the dataset (while dataset size remains constant), the size of the secondary structures increase. Consequently, the query time increases. In real life datasets, number of colors remain constant. So we used synthetic datasets $D1$ and $D2$ for observations. Average query time for BB k-d tree for both the datasets is increasing with the increase in the number of colors as shown in the Fig 4(a) and 4(b). However, in real life scenerio, the ratio of the number of colors in a dataset and the size of the dataset is generally very low.

6.3 Size of Query Box

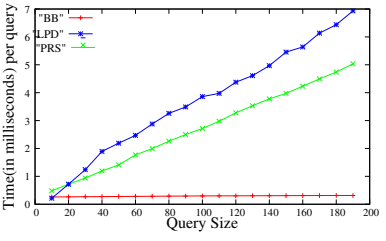
In general, for a given dataset, as the size of the query box increases, the number of points lying inside it also increases. So, naturally the query time of PRS and LPD techniques are expected to increase with increase in size of the query box. Interestingly, *BB k-d technique is minimally affected by the variation in the size of query box which is highly desirable.* As the query box size keeps increasing, the *depth* of the nodes being visited in the BB k-d tree decreases; since the existence or non-existence of the bounding box a color inside the query box becomes clear at an early stage. At the same time when the size of the query box is very small, then the number of primary structure nodes visited will also be less. Hence the query time is minimally varied w.r.t. the query box size. Experimental results have shown that with increase in size of the query box, initially the query time off BB k-d tree increaes slightly and then decreases or stays flat (see Fig 5). In 5(a) and 5(b), we observe the same pattern even when we vary the number of colors in the synthetic datasets $D1$ and $D2$.



(a) Uniform dataset D1

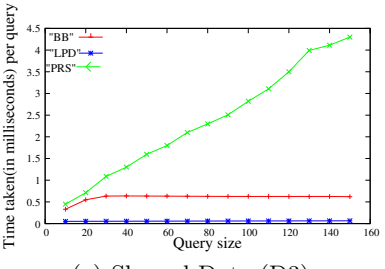


(b) Gaussian dataset D2

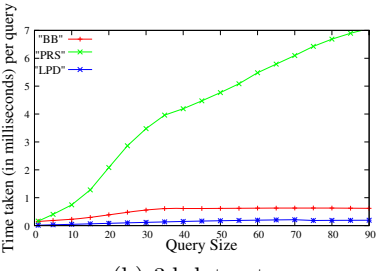


(c) Foreset Cover dataset R1

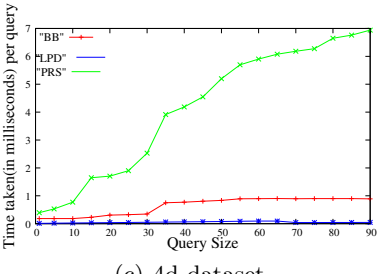
Fig. 5. Effect of size of query box on the query time. Two synthetic datasets and one real-life dataset have been used. In Fig 5(a) and 5(b) BB_x represents that the dataset used has x number of distinct colors. Note that the lines corresponding to BB k-d technique are almost flat.



(a) Skewed Data (D3)



(b) 3d-dataset



(c) 4d-dataset

Fig. 6. Effect of skewed datasets on query time. (b) and (c) are based on US census dataset (R2)

6.4 Effect of Data Distribution

Both PRS and BB k-d techniques use k-d tree as their primary structure. Hence, the distribution of data will have similar effect on both the data structures. On the other hand, LPD will perform well if the dataset is highly skewed w.r.t one of the dimensions. As described in Section 2, if the dataset is highly skewed, then the value of β will be small, resulting in good query performance. Fig 6(a) shows the results on the skewed synthetic dataset (D3) to validate our arguments. US Census data (Fig 6(b) and 6(c)) is skewed w.r.t *age* dimension and hence, LPD does well. However, the query time for BB k-d technique is always within a bounded factor of LPD's query time. However, in Fig 5(a) the datasets have uniform distribution which leads to poor perfomance of LPD.

7 Conclusions and Future Work

In this paper we came up with a main-memory data structure, BB k-d tree, to solve the orthogonal colored range searching problem (OCRS). In BB k-d tree, we augmented the traditional k-d tree with secondary data structures which resulted in significant improvement of the query response time (with minimal increase of $O(\log \alpha)$ factor in space). Comparision of this data structure was done with two base-line brute-force algorithms which solved the traditional orthogonal range searching problem. An existing theoretical solution was implemented but found unsuitable due to its high space consumption. Experiments were performed to compare our technique with the base-line techniques by varying factors such as number of colors (α), size of the query box and data distribution. The BB k-d tree performed consistently well under most of the conditions. In some minimal cases the base-line brute-force do better (in terms of query response) than BB k-d tree: In a highly skewed data LPD slightly performed better than BB k-d tree, PRS performs better than BB k-d tree in an extreme scenario where the number of colors in the dataset are almost close to the cardinality of the dataset. This BB k-d tree method is equally applicable to higher dimensions. Future work would involve coming with efficient execution plans for the query optimizer to answer OCRS query. There is need for practical solutions for aggregate queries on colored geometric problems [11,12] for main-memory, external memory models etc.

References

1. `http://www.mcobject.com/in_memory_database`
2. `http://www.ics.uci.edu/mlearn/MLRepository.html`
3. Agarwal, P.K., Govindarajan, S., Muthukrishnan, S.: Range Searching in Categorical Data: Colored Range Searching on Grid. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 17–28. Springer, Heidelberg (2002)
4. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In: ACM SIGMOD Conference, pp. 322–331 (1990)

5. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM 18, 509–517 (1975)
6. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational geometry: algorithms and applications. Springer, Heidelberg (2000)
7. Gupta, P., Janardan, R., Smid, M.: Further results on generalized intersection searching problems: counting, reporting, and dynamization. Journal of Algorithms 19, 282–317 (1995)
8. Gupta, P., Janardan, R., Smid, M.: Computational geometry: Generalized intersection searching. In: Mehta, D., Sahni, S. (eds.) Handbook of Data Structures and Applications, ch. 64, pp. 64-1–64-17. Chapman & Hall/CRC, Boca Raton, FL (2005)
9. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: SIGMOD Conference, pp. 47–57 (1984)
10. Janardan, R., Lopez, M.: Generalized intersection searching problems. International Journal on Computational Geometry & Applications 3, 39–69 (1993)
11. Rahul, S., Gupta, P., Rajan, K.: Data Structures for Range Aggregation by Categories. In: 21st Canadian Conference on Computational Geometry (CCCG 2009), pp. 133–136 (2009)
12. Rahul, S., Bellam, H., Gupta, P., Rajan, K.: Range aggregate structures for colored geometric objects. In: 22nd Canadian Conference on Computational Geometry (CCCG 2010), pp. 249–252 (2010)
13. Sellis, T.K., Roussopoulos, N., Faloutsos, C.: The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In: 13th International Conference on Very Large Data Bases (VLDB 1987), pp. 507–518 (1987)
14. Shi, Q., JáJá, J.: Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines. Information Processing Letters 95(3), 382–388 (2005)
15. Kumar, Y., Janardan, R., Gupta, P.: Efficient algorithms for reverse proximity query problems. In: GIS 2008, p. 39 (2008)