On k-d Range Search With Large k

by

Qingxiu Shi and Bradford G. Nickerson

TR06-176, May 31, 2006

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
www: http://www.cs.unb.ca

Abstract

We present two new k-dimensional data structures, called the PKD-tree and the PKD^+-tree, respectively. They are explored for indexing combined text and point data, and evaluated experimentally for orthogonal range search (for $2 \leq k \leq 100$ and n up to 1,000,000) using synthetic data and real data. We compared the range search performance of the PKD-tree with the PKD^+-tree, the k-d tree, the Pyramid technique, the R^*-tree and naive search. The experimental results show that the PKD-tree and the PKD^+-tree have very good performance for large k, and they always outperform the Pyramid technique, and are better than k-d tree and the R^*-tree when $k \geq \log_2 n$. For a PKD^+-tree built from n uniform and random data points, an orthogonal range search with a query square W of side length Δ visits $O(k \log n + n(1 - (1 - 2\Delta)^k))$ nodes for $\Delta \leq 0.5$.

Contents

List of Tables

List of Figures

1 Introduction

Multidimensional data are used in many applications: database applications, geographical information systems, computer graphics and computational geometry. Many applications require processing of large amounts of k-dimensional (k-d) data. The specific search problem we are concerned with is range search. Given a collection of n records, each containing k attributes, a range search asks for all records in the collection with key values inside a specified range for each of the k dimensions. Over the past 30 years, more than 100 data structures for the range search problem have been presented [1][4][6][7][8][12][15]. The motivation for this research is to find dynamic data structures that support efficient orthogonal range search for combined text and point data in low and high dimensions.

1.1 Related Work

The Patricia trie was discovered by D.R. Morrison [13]. Each partition of the Patricia trie splits a region of the search space into two equal subregions. Each coordinate axis gets cut in turn, in a cyclical fashion of $1, 2, \cdots, k, 1, 2, \cdots$. In [16] we compared the range search performance of Patricia tries with k-d trees and R^*-trees. When k is getting larger (e.g. $k \approx \log_2 n$), Patricia tries range search performance deteriorates. Table 1 shows the average height of Patricia tries built from n uniformly distributed k-d points ($n = 1,000,000$, and the bucket capacity is one). The height+skips means the height of trie plus the length of skipped strings stored in the internal nodes along the path from the root to the leaf node. We can see that the height+skips of the trie doesn't change with increasing k. When k=2, the space of each dimension can be divided in half 20 times on average from the root to the leaves, and nodes are pruned quickly during range search. When k=20, the space of each dimension is divided in half twice on average, which results in many more nodes visited during range search.

What's more, if we assume that the query rectangle W is a hypercube and the number of points F in range is fixed for the same n, then for uniformly and randomly distributed k-d points, we expect the side length Δ of the query hypercube is the k-th root of F/n. For example, for $n = 1,000,000$ and $E(F) = \log_2 n$, when k=2, $\Delta = (\log_2 n/n)^{1/2} \approx 0.0045$; when k=20, $\Delta = (\log_2 n/n)^{1/20} \approx 0.5821$, which is larger than half of each dimension in the search space. It is obvious that a range query with side length larger

Table 1: The average height of random Patricia tries (n=1,000,000).

	k=2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
height	26	26	26	26	26	26	26	26	27	26	26	27	26	27	27	26	26	26	26
height+skips	41	41	40	42	41	40	40	41	41	41	40	42	40	41	41	42	41	40	41

than 0.5 must intersect with the region with side length equals to or larger than 0.5 in every dimension, so we visit most of nodes in the Patricia trie during the range search (See Fig.1).

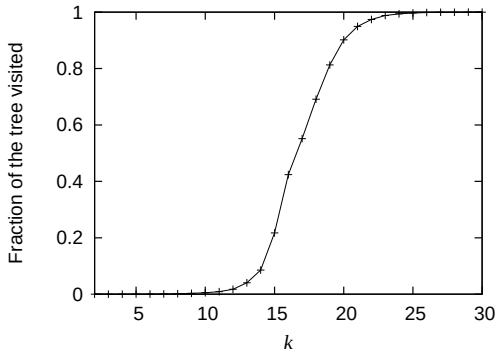


Figure 1: The fraction of the tree visited for the Patricia trie range search with k-d hypercubes, where $n = 1,000,000$, $2 \leq k \leq 30$ and $F = \log_2 n$.

The k-d tree is one of the original data structures for solving the point range search problem. The k-d tree was proposed by Bentley [3] as a generalization of the binary search tree in k-d space. At each intermediate node, the k-d tree divides k-d space into two parts by a (k-1)-dimensional hyperplane parallel to the coordinate axes. The direction of the hyperplane alternates between the k possibilities from one tree level to the next. The k-d tree performs efficiently on small k ($k \leq \log n$), however, when k is getting larger, the k-d tree is limited by the curse of dimensionality, i.e., the k-d tree tends to visit most of nodes in the tree during range search.

In recent years, several mapping-based indexing schemes have been proposed to improve the performance of range search in high-dimensional data space, e.g. the Pyramid technique [5] and the iMinMax(θ) [14]. Berchtold et al [5] showed that when processing range queries on large k, the perfor-

mance improves for increasing k. The basic idea is to transform the k-d data points into 1-d values, and then store and access the values using a B^+-tree. A k-d range query is mapped to a union of 1-d range queries. Based on the similarity between these schemes, Zhang et al. [17] proposed a generalized structure for multidimensional data mapping and query processing. The mapping-based indexing scheme overcomes the high dimensionality problem. However when $k \leq \log n$, the mapped-based indexing method tends to check more data points to decide if the data points in range than conventional indexing methods, e.g. the k-d tree and the Patricia trie.

1.2 Our Results

We combine the Pyramid technique and the k-d tree, called the PKD-tree and the PKD^+-tree, respectively, using the advantage of the k-d tree for small k ($k \leq \log_2 n$), and the superiority of the Pyramid technique for $k > \log_2 n$. In Section 2 we have a closer look at the Pyramid technique and its related work. In Sections 3 and 4, we proposed these two data structures, and theoretically analyzed their space requirement and range search time in the worst case. We present the experimental results from an extensive performance study to evaluate the PKD-tree and the PKD^+-tree for orthogonal range search in Section 5, using synthetic data and real data. We compare the range search performance of the PKD-tree to the PKD^+-tree, the k-d tree, the Pyramid technique, the R^*-tree [2] and naive search. Overall, our experiments show that the PKD-tree is greatly better than the Pyramid technique when $k \leq \log_2 n$, and it and its variant approximate the Pyramid technique when $k >> \log_2 n$. The PKD-tree and the PKD^+-tree outperform the k-d tree and the R^*-tree when $k \geq \log_2 n$ and don't deteriorate with increasing k. Without loss of generality, the following discussions are all based on unit space $[0, 1]^k$.

2 The Pyramid Technique and Related Work

2.1 Pyramid technique

The basic idea of the Pyramid technique is to transform the k-d point data into 1-d values, and then store and access the 1-d values using a B^+-tree. The data space is divided in two steps: firstly, the data space is split into

$2k$ pyramids having the center point of data space $(0.5, 0.5, \cdots, 0.5)$ as their top and a $(k-1)$-d surface of the data space as their base. Secondly, each of the $2k$ pyramids is divided into several partitions, each corresponding to one data block of the B^+-tree. Fig.2 shows a 2-d example. The data space is divided into 4 triangles, sharing the center point of the data space as their top and one edge of the data space as a base. Then these 4 partitions are split again into several data blocks parallel to the data boundary.

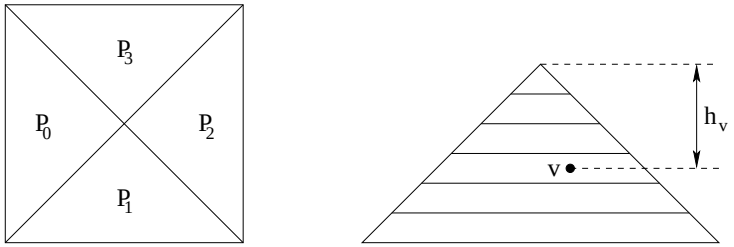


Figure 2: A 2-d Pyramid technique example.

A number i $(0 \leq i < 2k)$ is assigned to each pyramid according to some rule. Assume a point $v = (v_0, v_1, \cdots, v_{k-1})$ is in pyramid i. The height h_v of the point is defined to be the distance between v and the center in dimension $i \bmod k$, i.e. $h_v = |0.5 - v_{i \bmod k}| \in [0, 0.5]$. The pyramid value of v is defined as the sum of its pyramid number i and its height h_v: $pv_v = i + h_v$. The pyramid i covers an interval of $[i, i + 0.5]$ pyramid values and the sets of pyramid values covered by any two different pyramids are disjoint.

After determining the pyramid value of a point v, we insert v into a B^+-tree using pv_v as a key, and we store the point v in the corresponding leaf node of the B^+-tree. Two points may have the same pyramid value, but the points are stored in the leaf nodes, so we don't need an inverse transformation. We give a 2-d example in Fig.3.

Given a query rectangle W, the points in W are reported. In the first step, we determine which pyramids are intersected by W. In the second step, we determine which pyramid values inside an intersected pyramid p_i intersect W. Fig.4 shows the region visited when an orthogonal range search is performed. The black area is the region of the query rectangle W, and the cross-hatched area is the region we need to visit during the range search in addition to W.

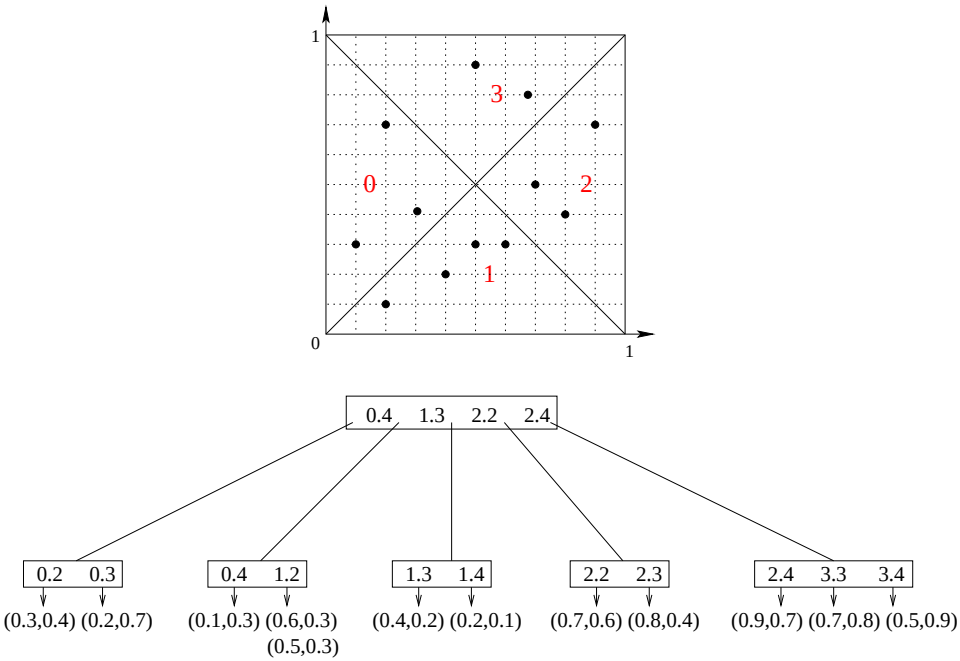


Figure 3: (above) A set of points in 2-dimensional data space (the numbers in the triangles are the pyramid numbers i), and (below) the corresponding B^+-tree (the maximum number of keys is 4, and the point insertion order is (0.2,0.7), (0.1,0.3), (0.3,0.4), (0.2,0.1), (0.4,0.2), (0.5,0.3), (0.6,0.3), (0.8,0.4), (0.7,0.5), (0.9,0.7), (0.7,0.8), (0.5,0.9)).

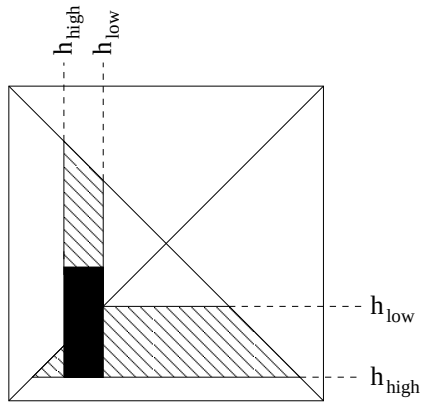


Figure 4: The data space and the query rectangle W (the black area is the region of W, and the cross-hatched area is the region needed to be visited during the range search in addition to W) .

2.2 The P^+-tree

Zhang and Ooi [18] noticed that the effectiveness of the Pyramid technique is sensitive to the positions of the query rectangle and the performance of the Pyramid technique is dependent on the distribution of the data set. They proposed the P^+-tree to avoid these problems: first, the data space is divided into subspaces using the hyperplane parallel to the coordinate axes (like the k-d tree), then each subspace is mapped into a hypercube so that the Pyramid technique can be applied in each subspace (See Fig.5). They showed that the P^+ has 2 to 5 times performance improvements over the Pyramid technique for orthogonal range search.

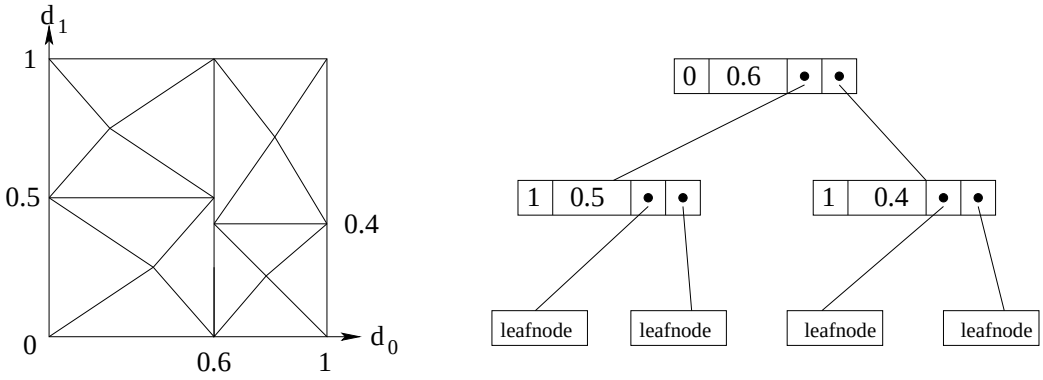


Figure 5: A P^+-tree example. The leaf node points to the corresponding pyramid.

3 The PKD-tree

Notice that if we divide each pyramid into subregions, e.g. as shown in Fig.6, the pyramid is divided by the dotted lines, and the ideal search space is the shadow area, which is a smaller region visited during range search compared to Fig.4. Based on this key observation, we propose the PKD-tree.

3.1 The Structure of PKD-tree

Given a k-d key $v = (v_0, v_1, \cdots, v_{k-1})$, the attribute v_i can be a point coordinate value or a text string, $0 \leq i \leq k-1$. When v_i is a text, we use the

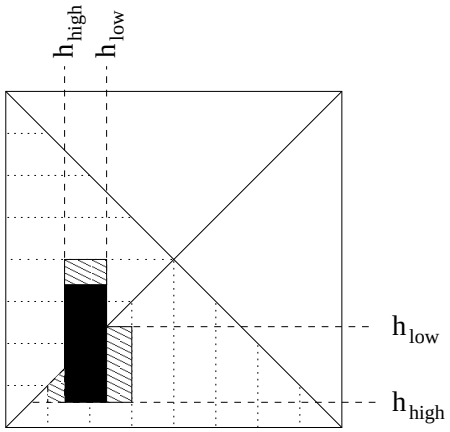


Figure 6: Data space and query rectangle W. The black area is the region of W, and the cross-hatched area is the region visited during the range search in addition to W.

numeric mapping method [11] to get a numeric value in $[0, 1]$ for v_i: assume the text data is comprised of symbols drawn from an alphabet of size α, and each symbol is mapped to an integer in the range 0 to α-1. Let a string of length c be $s_1 s_2 \cdots s_c$, with each symbol s_i mapped to an integer t_i, the string s is mapped to $\frac{t_1}{\alpha} + \frac{t_2}{\alpha^2} + \frac{t_3}{\alpha^3} + \cdots + \frac{t_c}{\alpha^c}$, which is a one-to-one mapping. We denote the k-d key v as $\overline{v} = (\overline{v}_0, \overline{v}_1, \cdots, \overline{v}_{k-1})$ after the numeric mapping, $\overline{v}_i = v_i$ if v_i is a point data, or $\overline{v}_i$ is the numeric value of v_i if v_i is a text data.

We say a key v located in pyramid p_i, $i = j_{max}$, if $(\overline{v}_{j_{max}} < 0.5)$; $i = k + j_{max}$, if $(\overline{v}_{j_{max}} \geq 0.5)$, where $j_{max} = (j | (\forall m, 0 \leq (j, m) < k, j \neq m : |0.5 - \overline{v}_j| \geq |0.5 - \overline{v}_m|))$ [5]. The distance from the point v to the center point of data space is defined to be $h_v = |0.5 - \overline{v}_{i \bmod k}|$, then the pyramid value $pv_v$of the point v is $i + h_{v_i \bmod k}$. The algorithm for calculating pv_v is given in Fig.7.

We denote by T the PKD-tree constructed by inserting n keys into an initially empty tree. The root of the PKD-tree is an internal node with $2k$ child pointers, indexing the $2k$ pyramids respectively. After determining which pyramid the key v is in, we insert v into the corresponding B^+-tree using the pyramid value pv_v as a key. When we reach the leaf level of the B^+-tree, we insert v into the associated k-d tree (See Fig.8). Let S be the maximum number of points in the k-d tree. If the number of points in the k-d tree is S before inserting v, the pyramid value pv_v is inserted into the

```
PyramidValue(Point v)
1  j_max ← 0; h_{j_max} ← |0.5 − v_0|
2  for (j = 1; j < k; j ← j + 1)
3  do if h_{j_max} < |0.5 − v_j|
4        then j_max ← j; h_{j_max} ← |0.5 − v_j|
5  if v_{j_max} < 0.5
6     then i ← j_max
7     else i ← k + j_max
8  pv_v ← i + h_{j_max}
9  return pv_v
```

Figure 7: Algorithm for calculating the pyramid value pv_v of a point v.

B^+-tree as a key, and the k-d tree is partitioned into two k-d trees according to the key values in the B^+-tree. As we know that the k-d tree is relatively slow for large k, we define $S = \sqrt{\frac{n}{2k}}$ such that each k-d tree has fewer points when k is large. This means more of the search is done in the pyramid part of the PKD-tree, which is more efficient for large k. After preprocessing all n keys, we obtain the tree T, which allows us to carry out an orthogonal range search. In the following discussions, we assume the B^+-tree has order M. The internal nodes of a B^+-tree of order M contains between M and $2M$ keys, and a node with m keys has $m + 1$ children.

Theorem 1 *The PKD-tree built from n k-d data points requires $O(kn)$ space.*

Proof. The root node of the PKD-tree requires $2k$ space. Assume B_i is the number of data points in the pyramid i, $0 \leq i \leq 2k-1$. There are at most $\frac{2B_i}{M}$ nodes in the B^+-tree indexing the pyramid i. So totally there are $\sum_{i=0}^{2k-1} \frac{2B_i}{M}$ nodes in the B^+-trees for all pyramids. Note that $\sum_{i=0}^{2k-1} B_i = n$, we have $\sum_{i=0}^{2k-1} \frac{2B_i}{M} = \frac{2n}{M}$. So the B^+-trees require $O(n)$ space. With the storage kn required for the k-d trees, the total storage is $O(kn)$. □

3.2 Orthogonal Range Search

Given a query rectangle $W = [L_0, H_0] \times [L_1, H_1] \times \cdots \times [L_{k-1}, H_{k-1}]$, the key $v = (v_0, v_1, \cdots, v_{k-1})$ is in the range iff $v_i \in [L_i, H_i]$, $\forall i \in (0, 1, \cdots, k-1)$.

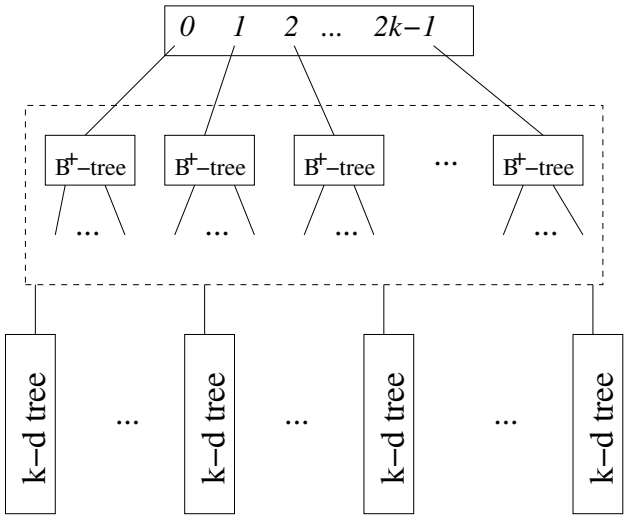


Figure 8: The data structure of the PKD-tree.

We define $\overline{W} = [\overline{L}_0, \overline{H}_0] \times [\overline{L}_1, \overline{H}_1] \times \cdots \times [\overline{L}_{k-1}, \overline{H}_{k-1}]$, where $\overline{L}_i = L_i - 0.5$ and $\overline{H}_i = H_i - 0.5$. A pyramid p_i is intersected by W if and only if

1. $\overline{L}_i \leq -MIN(\overline{L}_j, \overline{H}_j)$, if $i < k$, and
2. $\overline{H}_{i-k} \geq MIN(\overline{L}_j, \overline{H}_j)$, if $k \leq i < 2k$

$\forall j, 0 \leq j < k$, where $MIN(\overline{L}_j, \overline{H}_j) = 0$, if $\overline{L}_j \leq 0 \leq \overline{H}_j$, else $MIN(\overline{L}_j, \overline{H}_j) = min(|\overline{L}_j|, |\overline{H}_j|)$ [5]. The algorithm for calculating the pyramids intersected $\overline{W}$ is given in Fig. 9.

Then we need to calculate the interval $[h^i_{low}, h^i_{high}]$ that the pyramid values of all point data inside the intersection of W and pyramid p_i are in the interval $[i + h^i_{low}, i + h^i_{high}]$. We define a more restricted value of h_{low} than the one in [5].

The modified query rectangle $\widetilde{W}$ for pyramid i $\widetilde{W}_i = [\widetilde{L}_0, \widetilde{H}_0] \times [\widetilde{L}_1, \widetilde{H}_1] \times \cdots \times [\widetilde{L}_{k-1}, \widetilde{H}_{k-1}]$, where

1. $\widetilde{L}_j = \overline{L}_j$, $\widetilde{H}_j = min(\overline{H}_j, 0)$, if $i < k$ and $j = i \mod k$, or
2. $\widetilde{L}_j = max(\overline{L}_j, 0)$, $\widetilde{H}_j = \overline{H}_j$, if $k \leq i < 2k$ and $j = i \mod k$
3. $\widetilde{L}_j = \overline{L}_j$ and $\widetilde{H}_j = \overline{H}_j$ for $0 \leq j < k$, $j \neq i \bmod k$

```
INTERSECTION(W̄)
 1  Initialize boolean array intersect[2*k]←0
 2  for (i = 0; i < k; i ← i + 1)
 3  do x ← 0
 4     y ← 0
 5     for (j = 0; j < k and j ≠ i; j ← j + 1)
 6     do if (L̄_i ≤ (−MIN(L̄_j, H̄_j)))
 7           then x ← x + 1
 8        if (H̄_i ≥ MIN(L̄_j, H̄_j))
 9           then y ← y + 1
10     if (x = k − 1)
11        then intersect[i] ← 1
12     if (y = k − 1)
13        then intersect[k + i] ← 1
```

Figure 9: Algorithm determining which of $2k$ pyramids are intersected by $\overline{W}$.

Given a query rectangle $\overline{W}$ and an affected pyramid p_i, the intersection interval $[h_{low}, h_{high}]$ is define as follows:

1. $h^i_{low} = 0$, if $\overline{L}_j \leq 0 \leq \overline{H}_j$, $\forall j \in \{0, 1, \cdots, k-1\}$, or
2. $h^i_{low} = \bigwedge_{j \in \{0,\cdots,k-1\}} max(MIN(\widetilde{L}_{i \bmod k}, \widetilde{H}_{i \bmod k}), MIN(\widetilde{L}_j, \widetilde{H}_j))$
3. $h^i_{high} = max(|\widetilde{L}_{i \bmod k}|, |\widetilde{H}_{i \bmod k}|)$

$\bigwedge_{j \in \{0,\cdots,k-1\}} a_j$ is the minimum of the numbers a_j. The algorithm determining h_{low} and h_{high} is given in Fig.10.

Range search begins from the root of the PKD-tree. If the pyramid i ($0 \leq i \leq 2k-1$) is intersected by W (INTERSECTION algorithm in Fig.9), we visited the B^+-tree which the ith child the root points to using interval $[i + h^i_{low}, i + h^i_{high}]$. When we reach the last level of the B^+-tree, we visit the according k-d tree to determine if the points inside W: starting at the root of the k-d tree, at each node v, we compare the v_j attribute of the current node with $[L_j, H_j]$ ($0 \leq j < k$, j is the current node's discriminator). If $v_j \leq L_j$, the search continues on the left child of v; if $v_j > H_j$, the search continues on the right child of v; otherwise, we check if v inside W and the search continues on both children of v.

```
INTERVAL(W̄)
 1  for (i = 0; i < 2k; i ← i + 1)
 2  do if (i < k)
 3        then q_{i_min} ← L̄_i
 4             q_{i_max} ← min(H̄_i, 0)
 5        else q_{i_min} ← max(L̄_{i−k}, 0)
 6             q_{i_max} ← H̄_{i−k}
 7     m ← 0
 8     for (j = 0; j < k; j ← j + 1)
 9     do if (L̄_j ≤ 0) and (H̄_j ≥ 0)
10           then m ← m + 1
11     if (m = k)
12       then h_low[i] ← 0
13       else q_{j_max} ← 0
14            q_{j_min} ← 0.5
15            for (j = 0; j < k and j ≠ i; j ← j + 1)
16            do q_{j_max} = max(MIN(q_{i_min}, q_{i_max}), MIN(L̄_j, H̄_j))
17               if q_{j_min} > q_{j_max}
18                  then q_{j_min} ← q_{j_max}
19            h_low[i] ← q_{j_min}
20     h_high[i] ← max(|q_{i_min}|, |q_{i_max}|)
```

Figure 10: Algorithm for determining h^i_{low} and h^i_{high} in each pyramid i.

Theorem 2 *Given a PKD-tree built from n k-d data points drawn from a uniform random distribution* $[0,1]^k$*, and a query square* W *with side length* Δ*, an orthogonal range query visits*

1. $O(k \log \frac{n}{kS} + \frac{n(1-(1-2\Delta)^k)}{S^{1/k}} + F)$*, if* $\Delta \leq 0.5$
2. $O(k \log \frac{n}{kS} + \frac{n(1+(2\Delta-1)^k)}{S^{1/k}} + F)$*, if* $\Delta > 0.5$

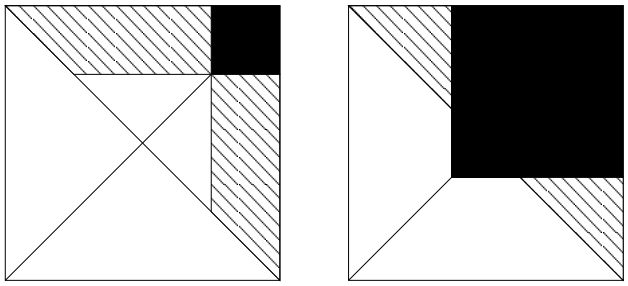
nodes in the PKD-tree, where F *is the number of data points in range.*

Proof. We assume the input data points are randomly and uniformly distributed. For an input size n, the number of data points in each B^+-tree is on average $\frac{n}{2k}$. Let S be the maximum number of nodes in the associated k-d tree, then the number of k-d trees in each B^+-tree of order M is on average

$\frac{n}{2kS}$, and the expected height of all B^+-trees of order M is $\log_{M+1}\frac{n}{2kMS}$. The worst case happens when W is in the corner of the data space (See Fig.11).

When $\Delta \leq 0.5$, the volume of pyramid i to be visited intersecting W is $\frac{1}{2k} - \frac{(2(0.5-\Delta))^k}{2k} = \frac{1}{2k}(1-(1-2\Delta)^k)$. There are $\frac{n}{2k}(1-(1-2\Delta)^k)$ data points in the shadow region of pyramid i. The number of k-d trees associated to the B^+-tree for the pyramid i to be visited is $\lceil \frac{n}{2kS}(1-(1-2\Delta)^k) \rceil$. The worst case of the number of nodes visited in the k-d tree having S nodes during range search is $O(S^{(1-1/k)}+f)$, where f is the number of nodes in range in the k-d tree. The number of leaf nodes visited in the B^+-tree is $\lceil \frac{\lceil \frac{n}{2kS}(1-(1-2\Delta)^k) \rceil}{M} \rceil$, and the number of internal nodes visited is the height of the B^+-tree. There are k pyramids intersecting W, so the total nodes visited in the PKD^+-tree is thus $O(k\log_{M+1}\frac{n}{kMS} + \frac{n(1-(1-2\Delta)^k)}{S^{1/k}} + F)$.

In a similar way, when $\Delta > 0.5$, the center of the data space is contained in W, so all pyramids are intersected by W. The total volume to be visited is $k(\frac{1}{2k} + \frac{(2(\Delta-0.5))^k}{2k}) = \frac{1}{2}(1+(2\Delta-1)^k)$, The maximum number of nodes visited in the PKD^+-tree is thus $O(k\log_{M+1}\frac{n}{kMS} + \frac{n(1+(2\Delta-1)^k)}{S^{1/k}} + F)$. □



$\Delta \leq 0.5$ $\Delta > 0.5$

Figure 11: The data space and the query square W (the black area is the region of W, and the cross-hatched area is the region needed to be visited during the range search in addition to W).

Theorem 3 *The expected number of nodes visited for an orthogonal range search in the PKD-tree built from n uniform and random k-d data points is $\Omega(k\log\frac{n}{kS} + F)$, where F is the number of data points in range.*

Proof. Given a query square W with side length Δ, the lower bound of the number of nodes visited during an orthogonal range search happens when the center of W is the center of the data space. All the $2k$ pyramids intersect W. In this case, we don't need to search extra region except the region of W. As illustrated in Fig.12, the region of the space needed to be visited is Δ^k, we have $F = \Delta^k$. As the expected height of each B^+-tree of order M is $\log_{M+1} \frac{n}{2kMS}$, the total number of nodes visited in the PKD-tree is $2k \log_{M+1} \frac{n}{2kMS} + F$. □

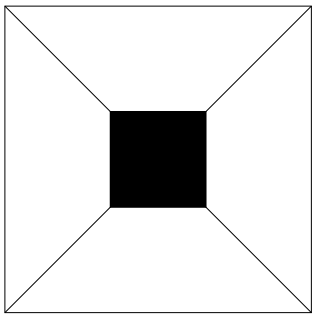


Figure 12: The data space and the query square W for the best case orthogonal range search (the black area is the region of W, which is the region needed to be visited during the range search).

4 The PKD^+-tree

The challenge for the PKD-tree is to determine an appropriate S in the k-d trees such that an excellent range search performance of the PKD-tree is always achieved. We arrange the k-d tree and the B^+-tree in another way, called the PKD^+-tree.

The basic structure of the PKD^+-tree is a B^+-tree of order M. The pyramid values of the points data are used as the key value in the nodes of the B^+-tree. An internal node with m keys of the PKD^+-tree has one right pointer, $m + 1$ child pointers and $m + 1$ associated k-d tree pointers, each child pointer is related to a k-d tree (denoted as KD) pointer. The data points stored in the leaf nodes of the subtree which the child pointer pointed to are store in the nodes of the k-d tree (e.g. in Fig. 13, the leftmost KD of the root contains (0.3,0.4) and (0.2,0.7), and the rightmost KD has (0.9,0.7),

(0.7,0.8) and (0.5,0.9)). The leaf node of the PKD-tree with m keys has one right pointer and m data point pointers. The right pointer points to the immediate right node at the same level.

Theorem 4 *The PKD^+-tree of order M built from n k-d data points requires $O(kn \log n)$.*

Proof. The height of the B^+-tree of order M is $O(\log_{M+1} n)$. The number of the nodes in the B^+-tree is $O(\frac{n}{M})$, which require $O(n)$ spaces. The leaf nodes need additional kn space for the data points. At each level above the leaf node level, there are totally n nodes in all associated KDs, which require kn spaces. With the storage required for the B^+-tree, the total storage is $O(kn \log_{M+1} n)$. □

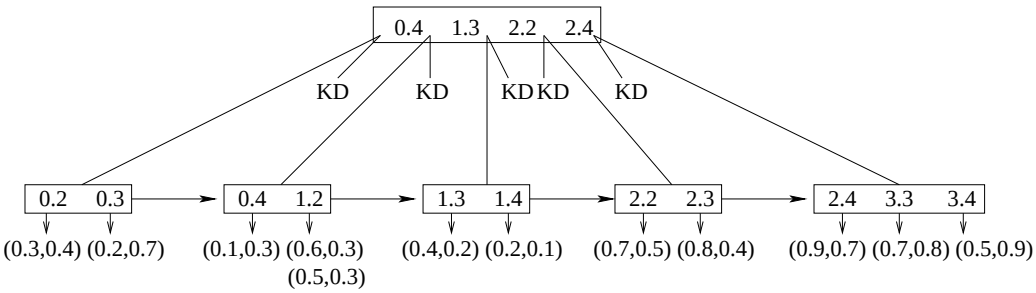


Figure 13: A 2-d PKD^+-tree example.

Range search algorithm is given in Fig.14. h^i_{low} and h^i_{high} are the same as the ones in the PKD-tree range search.

Theorem 5 *Given a PKD^+-tree build from n k-d data points drawn from a uniform random distribution $[0,1]^k$, and a query square W with side length Δ, an orthogonal range search query visits*

1. $O(k \log n + n(1-(1-2\Delta)^k))$, *if* $\Delta \leq 0.5$
2. $O(k \log n + n(1+(2\Delta-1)^k))$, *if* $\Delta > 0.5$

nodes in the PKD^+-tree.

```
RangeSearch(W)
 1  A ← empty set  // the set of points intersecting W
 2  l ← the height of the B+-tree
 3  for (i = 0; i ≤ 2k − 1; i ← i + 1)
 4  do if (intersect[i] = 1)
 5        then j ← 0
 6             m ← the number of keys in the node t
 7             t.key[s] ← the largest key ≤ i + h^i_low in the root node
 8             while (s < m and j < l)
 9             do if (s < m − 1)
10                   then if ([t.key[s], t.key[s + 1]] ∈ [i + h^i_low, i + h^i_high])
11                             then KDS(t.kd[s], W, A)
12                                  s ← s + 1
13                             else t ← t.child[s]
14                                  s ← 0
15                                  j ← j + 1
16                   else s ← 0
17                        if (t.right ≠ null)
18                          then t_right ← t.right
19                               if ([t.key[m − 1], t_right.key[s]] ∈ [i + h^i_low, i + h^i_high])
20                                 then KDS(t.kd[m], W, A)
21                                      KDS(t_right.kd[s], W, A)
22                                      t ← t_right
23                                 else t ← t.child[m]
24                                      j ← j + 1
25                          else t ← t.child[m]
26                               j ← j + 1
27             if (j = l)
28               then while (t ≠ nil and t.key[0] ≤ i + h^i_high)
29                      do s ← 0
30                         while (s < m and t.key[s] ≤ i + h^i_high)
31                         do if (t.key[s] ∈ [i + h^i_low, i + h^i_high])
32                               then Check(t.point[s], W, A)
33                            s ← s + 1
34                         t ← t.right
```

Figure 14: Orthogonal range search algorithm for the PKD^+-tree.

Proof. The worst case happens when the range search doesn't search any KD and checks the data points in leaf nodes, and W is in the corner of the data space, sharing a vertex and k edges with the space (See Fig.11). The height of the B^+-tree of order M is at most $\lfloor \log_{M+1} n \rfloor$.

When $\Delta \leq 0.5$, the volume of pyramid i to be visited intersecting W is $\frac{1}{2k} - \frac{(2(0.5-\Delta))^k}{2k} = \frac{1}{2k}(1-(1-2\Delta)^k)$. There are k pyramids intersecting W, so the total volume to be visited is $\frac{1}{2}(1-(1-2\Delta)^k)$. For uniformly and randomly distributed points, there are at most $\frac{1}{2}n(1-(1-2\Delta)^k)/M$ leaf nodes in the B^+-tree, and $\frac{1}{2}n(1-(1-2\Delta)^k)$ data points pointed by the leaf nodes visited. With the number of internal nodes visited in the B^+-tree, the maximum number of nodes visited in the PKD^+-tree is thus $k \log_{M+1} n + \frac{M+1}{2M} n(1-(1-2\Delta)^k)$.

When $\Delta > 0.5$, the center of the data space is contained in W, so all pyramids are intersected by W. The total volume to be visited is $k(\frac{1}{2k} + \frac{(2(\Delta-0.5))^k}{2k}) = \frac{1}{2}(1+(2\Delta-1)^k)$, so there are at most $\frac{1}{2}n(1+(2\Delta-1)^k)/M$ leaf nodes in the B^+-tree, and $\frac{1}{2}n(1+(2\Delta-1)^k)$ data points pointed by the leaf nodes visited. The maximum number of nodes visited in the PKD^+-tree is thus $2k \log_{M+1} n + \frac{M+1}{2M} n(1+(2\Delta-1)^k)$. □

Theorem 6 *The expected number of nodes visited for an orthogonal range search in the PKD^+-tree built from n uniform and random k-d data points is $\Omega(k \log n + F)$, where F is the number of data points in range.*

Proof. Given a query square W with side length Δ, the lower bound of the number of nodes visited during an orthogonal range search happens when the center of W is the center of the data space. In this case, we don't need to search extra region except the region of W. As illustrated in Fig.12, the region visited is Δ^k, we have $F = \Delta^k$. All $2k$ pyramids intersect W. As the expected height of the B^+-tree of order M is $\log_{M+1} n$, the total number of nodes visited in the PKD^+-tree is $2k \log_{M+1} n + F$. □

5 Experiments

We have conducted a series of experiments to evaluate the range search performance of the PKD-tree. Our experiments were performed using uniformly and randomly distributed data points from the interval $[0, 1]^k$, a color histogram data set [10] and the text data which are the name strings randomly chosen from the Canadian geographical names database [9], for $2 \leq k \leq 100$, and n up to 1,000,000. The programs were written in C++, and run on a Sun Microsystems V60 with two 2.8 GHz Intel Xeon processors and 3 GB main memory. Each experimental point in the following graphs was done with an average of 300 test cases.

5.1 Pyramid Technique

The number of nodes visited in the B^+-tree includes two parts: the number of pages accessed in the B^+-tree and the number of data points visited whose $i + h_{low} \leq pv_v \leq i + h_{high}$ when we reach the leaf node of the B^+-tree. We built the B^+-tree of order $M$with different values of M. The number of pages accessed and the points visited comparison between different M is shown in Fig.15, where the number of the set of data points $n = 1,000,000$, and the expected number of points in range $E(F) = \log_2 n$. There is a great difference between the number of pages accessed due to different M, but the number of data points visited is the same because the number of data points whose pyramid value in the interval $[i+h_{low},i+h_{high}]$ depends on the query rectangle only, independent of the B^+-tree. In the following discussion, the number of nodes visited using the Pyramid technique is the sum of the number of nodes visited in the B^+-tree and the number of data points pointed visited by the leaf nodes in the B^+-tree. For our testing, we used a value of M=5 for the B^+-tree.

5.2 Synthetic Uniform Point Data

In this section, we considered the input data are k-d points which were uniformly and randomly generated. We compared the number of nodes visited and the time taken to perform the range search in the PKD-tree with the Pyramid technique, the k-d tree and the naive search. We assumed the whole PKD-tree resides in the main memory, and there is no I/O disk access.

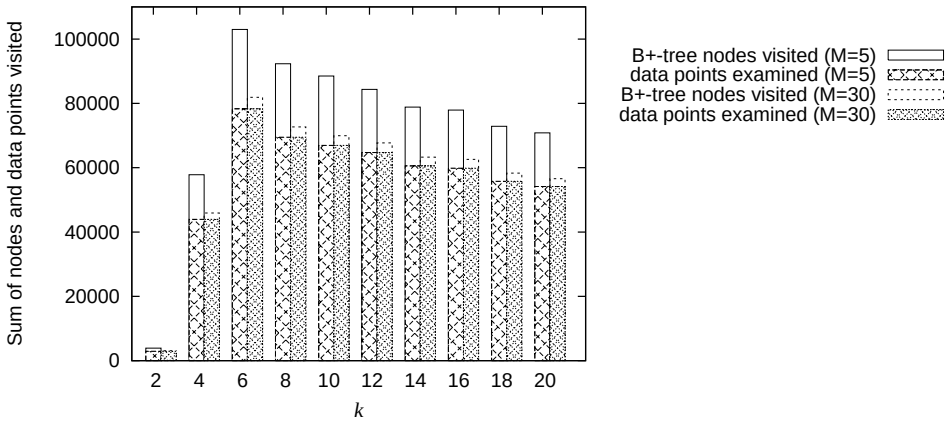
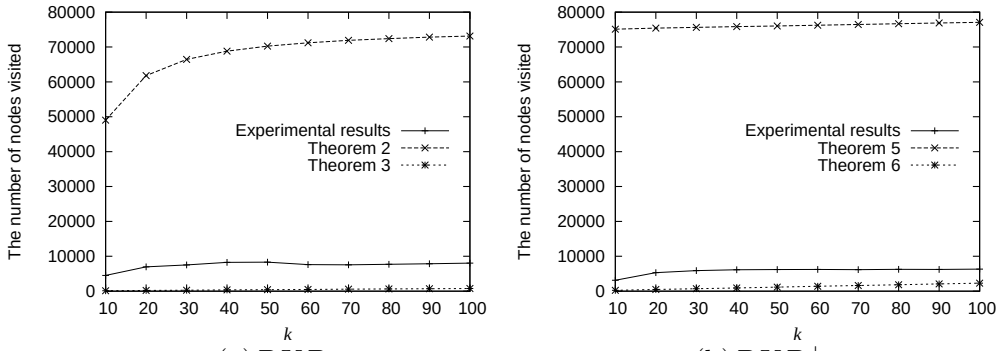


Figure 15: The sum of the number of nodes visited and the number of data points visited from the leaf nodes in the B^+-tree during range search for different values of M, where $E(F) = \log_2 n$, $2 \leq k \leq 20$ and $n = 1,000,000$. Note that the number of data points visited is independent of M.

5.2.1 Effects of k

To determine the influence of the dimension on the range search performance, we varied k from 2 to 100. We fixed the input data points size $n = 100,000$, and the output $F = \log_2 n$. Theorems 2 and 5 give the upper bound of the number of nodes visited when performing an orthogonal range search for the PKD-tree and the PKD^+-tree, respectively, and Theorems 3 and 6 give the lower bounds. The comparison between the experimental number of nodes visited and the theoretical results for the PKD-tree and the PKD^+-tree is shown in Fig.16. For the constant F, the query rectangle varies according to k. The experimental results in Fig.17 show that the PKD-tree and the Pyramid technique don't deteriorate with the increment of k, but the k-d tree suffers from the curse of dimensionality. When $k \leq 11$, in terms of the range search time spent, the PKD-tree is a little worse than the k-d tree, but great better than the Pyramid technique, with a speed-up factor up to 60; when $k > 11$, the PKD-tree is much better than the k-d tree, and the range search performance of the PKD-tree approximates the Pyramid technique when k approximates 100.



(a)PKD-tree (b)PKD^+-tree

Figure 16: The experimental number of nodes visited and the theoretical results, where $E(F) = \log_2 n$, $10 \leq k \leq 100$ and $n = 100,000$. The large difference is due to the theoretical results corresponding to upper bounds and lower bounds.

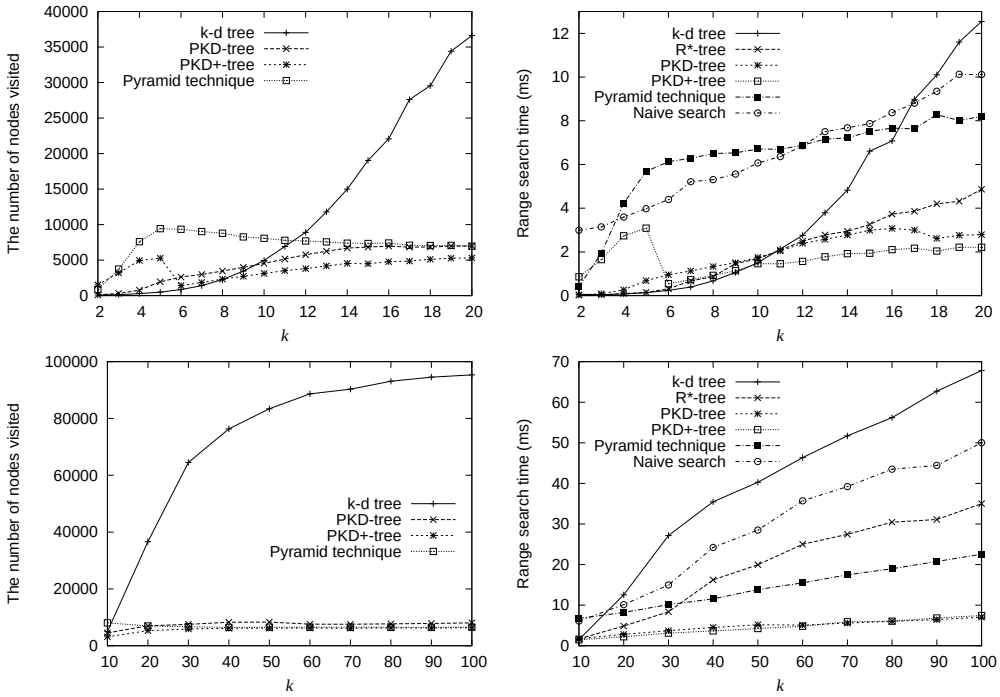


Figure 17: The experimental number of nodes visited and the range search time comparison between the PKD-tree, the PKD^+-tree, the k-d tree, the R^*-tree, the Pyramid technique and the naive search ($n = 100,000$, (above) $2 \leq k \leq 20$, (below) $20 \leq k \leq 100$ and $E(F) = \log_2 n$).

5.2.2 Effects of n

In these experiments, we measured the range search performance behavior with n varying from 100,000 to 1,000,000. The number of data points in range F is set to be $(\log_2 n)^2$, and the dimension k is 16 in Fig.18 For the number of nodes visited during the range search, the PKD-tree has a speed-up factor over the k-d tree up to 2.9, and a speed-up factor over the Pyramid technique up to 2. In terms of the range search time spent, the PKD-tree is up to 3.9 times faster than the k-d tree, 3.6 times faster than the Pyramid technique, and 2.4 times faster than the R^*-tree.

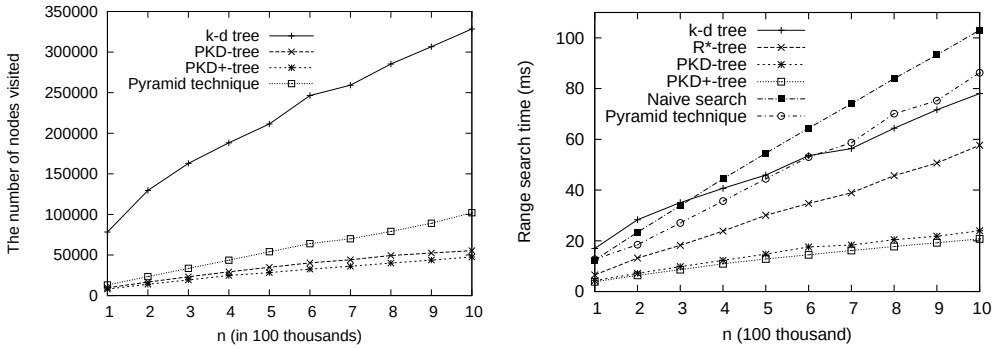


Figure 18: The experimental number of nodes visited and the range search time comparison between the PKD-tree, the k-d tree, the Pyramid technique and the naive search, where $E(F) = (\log_2 n)^2$, $100,000 \leq n \leq 1,000,000$ and $k = 16$.

5.2.3 Effects of F

We varied F from $0.00001n$ to $0.1n$. The size of data points $n = 1,000,000$, and the dimension $k = 20$. The experimental results are shown in Fig.20. It is reasonable that as F increases, the number of nodes visited and the range search time become larger. The PKD-tree outperforms the k-d tree and the Pyramid technique tree in both the number of nodes visited and the range search time for different F. For the number of nodes visited, the PKD-tree has a speed-up factor up to 5.7 over the k-d tree and 1.8 over the Pyramid technique. In terms of the range search time spent, the PKD-tree is up to 6.6 times faster than the k-d tree, 5.1 times faster than the Pyramid technique and 8.2 times faster than the R^*-tree.

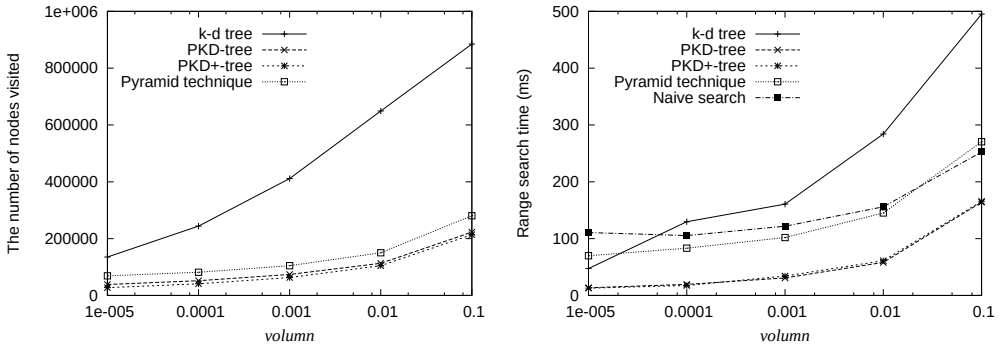


Figure 19: The experimental number of nodes visited and the range search time comparison between the PKD-tree, the k-d tree, the Pyramid technique and the naive search, where $volumn = E(F)/n$, $n = 1,000,000$ and $k = 20$.

5.3 Real Data

We tested data structures on color histogram data set [10], which has 68,040 32-dimensional data points on $[0,1]^{32}$. In Fig.20, the volume of the query square W with side length Δ ($volume$=Δ^d=$E(F)/n$) is varied from 0.0001 to 0.1. The PKD-tree has a speed-up factor between 2.4 ($volume$=0.1) and 16 ($volume$=0.0001) compared to the k-d tree, and between 1.1 and 3.4 compared to the Pyramid technique. The PKD-tree is slightly faster than the PKD^+-tree.

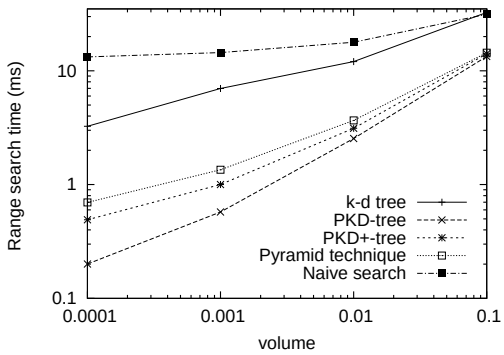


Figure 20: $volume$=$E(F)/n$, n=68,040 and d=32.

5.4 Combined Text and Point Data

In this section, the point data are randomly and uniformly generated, and the text data are the name strings randomly chosen from the Canadian geographical names database [9]. We used kr to denote the number of the dimensions of the text data, i.e. given a k-d key $v = (v_0, v_1, \cdots, v_{k-1})$, $kr = 2$ means that 2 attributes in v (any two in v) are the text data, and the rest $k-2$ attributes are the point data. We tested the range search performance of the PKD-tree with different k and F as what we did in the last section, and compared the PKD-tree to the k-d tree, the Pyramid technique and naive search.

From the experimental results shown in Fig.22, when $E(F) = \log_2 n$ and $n = 1,000,000$, in terms of the range search time spent, when $k \leq 10$, the PKD-tree is a little worse than the k-d tree, but with the increment of k, the PKD-tree is up to 8 times faster than the k-d tree. The PKD-tree is always better than the Pyramid technique, and it runs 7 to 56.8 times faster than the Pyramid technique. The difference is getting smaller when k is getting larger. When the expected output size $E(F) = (\log_2 n)^2$, the k-d tree is at most 2 times faster than the PKD-tree for $k \leq 7$, however the PKD-tree is 1.2 to 11.2 times faster than the k-d tree when $k > 7$. With the increment of k, the ratio between the k-d tree and the PKD-tree is getting larger. The PKD-tree is always better than the Pyramid technique, and it is 3.3 to 27.5 times faster than the Pyramid technique. Fig.21 shows that the PKD-tree is always the best and up to 13.6 times faster than the k-d tree and 8.5 times faster than the Pyramid technique in terms of the range search time.

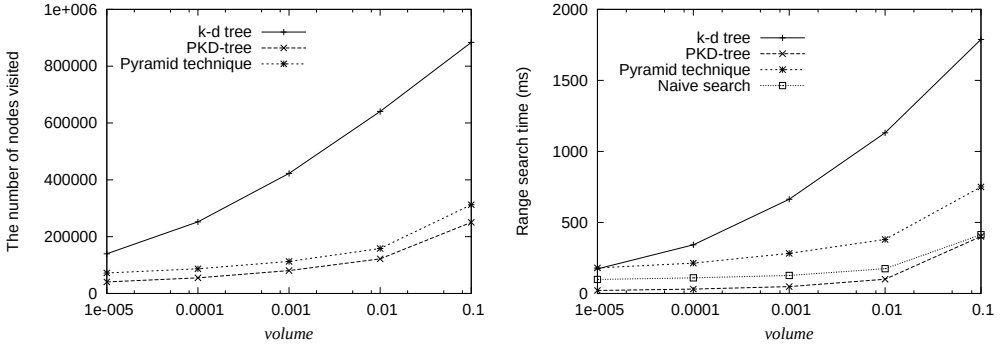


Figure 21: Comparisons between the PKD-tree, the k-d tree, the Pyramid technique and the naive search for combined text and point data, where $volume = E(F)/n$, $n = 1,000,000$, $k = 20$ and $kr = 2$.

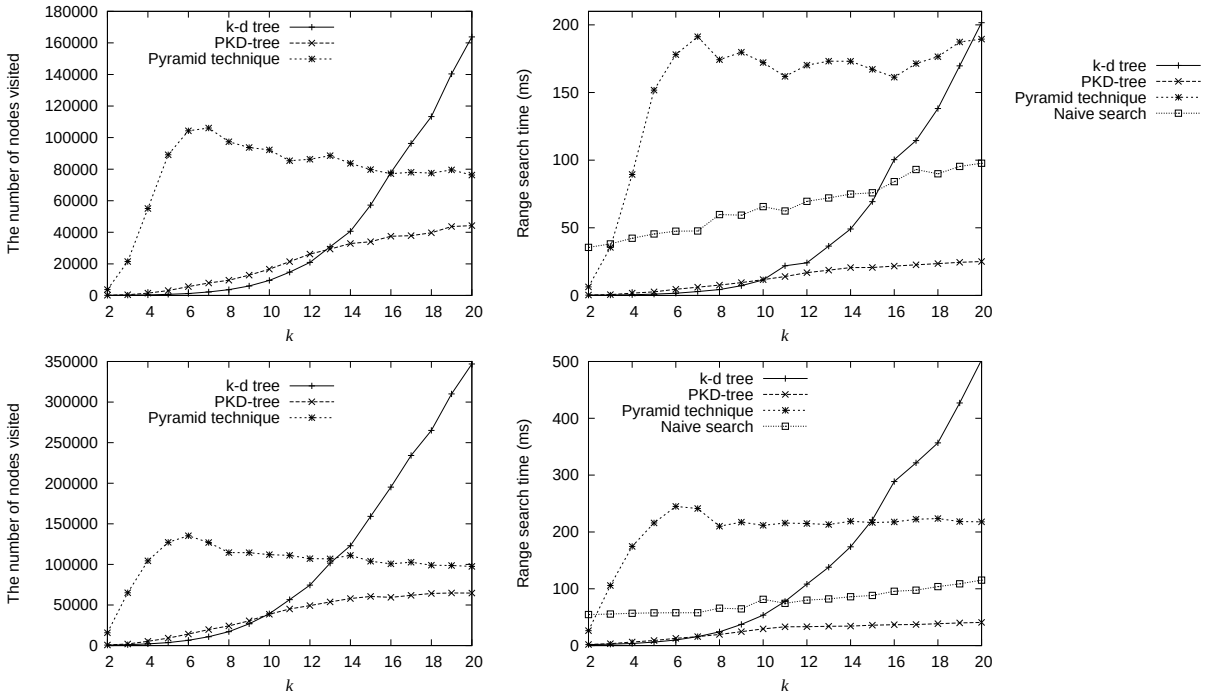


Figure 22: The experimental number of nodes visited and the range search time comparison between the PKD-tree, the k-d tree, the Pyramid technique and the naive search for combined text and point data. (above) $E(F) = \log_2 n$, and (below) $E(F) = (\log_2 n)^2$ ($n = 1,000,000$, $2 \leq k \leq 20$ and $kr = 2$).

6 Conclusions and Future Work

We propose two new data structures for orthogonal range search in high dimensional data space, called the PKD-tree and the PKD^+-tree, respectively. They combine the Pyramid technique and the k-d tree, using the advantage of the k-d tree in low dimension ($k \leq \log n$), and the superiority of the Pyramid technique in high dimensional data spaces. We conducted an extensive experimental study to evaluate the range search performance of the PKD-tree and the PKD^+-tree, and compared them to the k-d tree, the R^*-tree, the Pyramid technique and naive search, using uniform randomly generated point data, real data [10] and place names selected from the Canadian names database [9]. Overall, the experimental results show that the PKD-tree and its variant work well for any value of k ($2 \leq k \leq 100$), and

they are always better than the Pyramid technique, and outperform the k-d tree and the R^*-tree when $k \geq \log_2 n$. The challenge for the PKD-tree are to determine an appropriate maximum number of nodes in each associative k-d tree such that an excellent range search performance of the PKD-tree is always achieved. What is the expected range search time of the PKD-tree and the PKD^+-tree?

References

[1] P. Agarwal. *Handbook of Discrete and Computational Geometry*, chapter Range Searching, pages 575–598. CRC Press LLC, Boca Raton, FL, 1997.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, Atlantic City, NJ, 1990.

[3] J. Bentley. Multidimensional binary search trees for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] J. Bentley and J. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, December 1979.

[5] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 142–153, Seattle, Washington, USA, 1998.

[6] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces–index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[7] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[8] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, 1998.

[9] GeoBase. Homepage: http://www.geobase.ca, 2004.

[10] S. Hettich and S. D. Bay. *The UCI KDD Archive*. http://kdd.ics.uci.edu, Department of Information and Computer Science, University of California, Irvine, CA, 1999.

[11] H. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 403–414, Dallas, Texas, USA, May 2000.

[12] D. Knuth. *The art of computer programming: sorting and searching*, volume 3, pages 492–512. Addison-Wesley, Reading, Mass., 2 edition, 1998.

[13] D. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 14(4):514–534, October 1968.

[14] B. C. Ooi, K.-L. Tan, C. Yu, and S. Bressan. Indexing the edges - a simple and yet efficient approach to high-dimensional indexing. In *19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 166–174, Dallas, Texas, USA, May 2000.

[15] H. Samet. *The design and analysis of spatial data structures.* Computer Science Department, University of Maryland, College Park, Maryland, 2004.

[16] Q. Shi and B. G. Nickerson. k-d range search with binary patricia tries. Technical report, TR04-168, Faculty of Computer Science, University of New Brunswick, December 2004, 35 pages.

[17] R. Zhang, P. Kalnis, B. C. Ooi, and K.-L. Tan. Generalized multidimensional data mapping and query processing. *ACM Transactions on Database Systems*, 30(3):661–697, September 2005.

[18] R. Zhang, B. C. Ooi, and K.-L. Tan. Making the pyramid techique robust to query types and workloads. In *Proceedings of the 20th international conference on data engineering*, pages 313–324, Washington, DC, USA, 2004. IEEE computer society.