Making the Pyramid Technique Robust to Query Types and Workloads

Rui Zhang Beng Chin Ooi Kian-Lee Tan
Department of Computer Science
National University of Singapore, Singapore 117543
{zhangru1, ooibc, tankl}@comp.nus.edu.sg

Abstract

The effectiveness of many existing high-dimensional indexing structures is limited to specific types of queries and workloads. For example, while the Pyramid technique and the iMinMax are efficient for window queries, the iDistance is superior for kNN queries. In this paper, we present a new structure, called the P^+-tree, that supports both window queries and kNN queries under different workloads efficiently. In the P^+-tree, a B^+-tree is employed to index the data points as follows. The data space is partitioned into subspaces based on clustering, and points in each subspace are mapped onto a single dimensional space using the Pyramid technique, and stored in the B^+-tree. The crux of the scheme lies in the transformation of the data which has two crucial properties. First, it maps each subspace into a hypercube so that the Pyramid technique can be applied. Second, it shifts the cluster center to the top of the pyramid, which is the case that the Pyramid technique works very efficiently. We present window and kNN query processing algorithms for the P^+-tree. Through an extensive performance study, we show that the P^+-tree has considerable speedup over the Pyramid technique and the iMinMax for window queries and outperforms the iDistance for kNN queries.

1. Introduction

Multidimensional data are common in many applications. These applications have a wide range of needs. Window queries[1] are common in CAD, medical image [3] and GIS systems. For example, the *Forest CoverType* data set [1] has 54 attributes, out of which 10 are quantitative. To analyze the data, (partial) window queries that specify some ranges of elevation, aspect, slope, cover type, etc are typical. KNN queries are common in content-based retrieval systems [10, 12].

While a large number of indexing techniques have been proposed to improve performance, most of these techniques are not sufficiently robust for a wide range of queries. For example, the Pyramid technique [7] and iMinMax [15] are efficient for window queries, but perform less satisfactorily for kNN queries. On the other hand, metric-based schemes such as the iDistance [21] are usually superior for kNN queries, but may not be usable for window queries. Moreover, these schemes typically perform well for certain workloads (data set size, dimensionality, data distribution, etc) and become inferior to sequential scan in other cases.

In this paper, we propose an index structure, called the P^+-tree, that supports both window and kNN queries under different workloads (different data set size and dimensionality, different data distribution, queries with different selectivities and different shapes) efficiently. Our scheme is based on the Pyramid technique, which is primarily designed for hypercube-shaped window queries (where the query rectangle has equal sides in all dimensions). While the Pyramid technique is shown to be much better than the Hilbert R-tree [13] and the X-tree [16] for data sets of dimensionality larger than 8, it has three deficiencies: its performance is sensitive to the positions of the query hypercube; it is less effective for clustered data sets; and it is inferior for partial window queries. In these cases, a sequential scan over the data set may be more effective.

The basic structure of the P^+-tree is essentially a B^+-tree that indexes subspaces of points under a new transformation method. The data space is first partitioned into subspaces based on clustering. Next, points in each subspace are mapped onto a single dimensional space using the Pyramid technique, and stored in the B^+-tree. To discriminate the points within each cluster, they are transformed into non-overlapping regions in the single dimensional space. The crux of the scheme lies in the choice of the transformation that has two crucial properties. First, it maps each subspace into a hypercube so that the Pyramid scheme can

[1] In the literature, the term "range query" has been used to mean window query (hyperrectangle shaped) and similarity range query (hypersphere shaped). To avoid ambiguity, we use the term "window query" instead of "range query" throughout this paper. The "range query" in [7] and [15], in fact, means window query.

be applied. Second, it shifts the cluster center to the top of the pyramid, which is the case that the Pyramid technique works very efficiently.

The rest of the paper is organized as follows: Section 2 reviews some related work. In section 3, we examine the Pyramid technique, identify its deficiencies, and discuss how to overcome them. Then we present the P^+-tree in section 4 and query processing schemes in section 5. In section 6, we present the results of a performance study of the P^+-tree. Section 7 concludes this paper.

2 Related Work

The R-tree [11] and its variations such as the R*-tree [4] and the X-tree [16] were first proposed to manage multi-dimensional data. They work well with low-dimensional data. But their performance deteriorates as the dimensionality increases, and become unacceptable in high-dimensional space due to large amount of overlap. This phenomenon is called the *curse of dimensionality*.

Indexing methods based on transformation were proposed to make window query processing in medium- and high-dimensional space more efficient. Among them, the Pyramid technique [7] is notable. In the Pyramid technique, the high-dimensional data points are transformed into one-dimensional values and the transformed values are indexed by the classic B^+-tree. Experiments show that the Pyramid technique outperforms the Hilbert R-tree [13] and the X-tree [16]. However, the Pyramid technique is primarily designed for hypercube shaped queries over uniform data. The poorer performance to handle other cases limits its usefulness in real applications.

The iMinMax [15] is another indexing scheme intended for high-dimensional window query based on a different transformation method. θ in the iMinMax is a tuning parameter which makes it adaptive to data distributions. As such, it performs better than the Pyramid technique when the data is skewed. Since θ is a global parameter, the iMinMax works well when the number of natural clusters is small.

On kNN queries, the SS-tree [20], SR-tree [14] and X-tree were designed to reduce the effects of high dimensionality by using different page regions or bigger node size. The M-tree [8] was proposed for the generic metric space. The IQ-tree [6] compresses the leaf nodes and the A-tree [17] uses virtual bounding rectangles to approximate data objects. However, according to the analysis in [19], sequential scan may be the best method especially for uniform data in very high-dimensional space. And the VA-file was proposed to accelerate sequential scan by vector approximation. More recently, the iDistance [21] adopts the transformation strategy on kNN search and BOND [9] uses vertical fragmentation to reduce the I/O cost of frequently observed query patterns.

3 A Closer Look at the Pyramid Technique

The Pyramid technique divides the d-dimensional data space into $2d$ pyramids that share the center point of the space as their top, and the (*d-1*)-dimensional surfaces of the space are their bases (Figure 1). Each pyramid has a pyramid number i according to some rule. The distance between a point v and the center in dimension i (or $i - d$ if $i \geq d$) is defined as the height of the point, h_v. Then, the pyramid value of v is defined as the sum of its pyramid number i and its height h_v.

$$pv_v = (i + h_v)$$

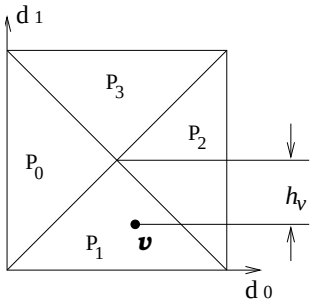


Figure 1. The Pyramid technique

This pyramid value is the key indexed by a B^+-tree. A query rectangle corresponds to a height range in an intersected pyramid. Those data points of height within the height range are accessed.

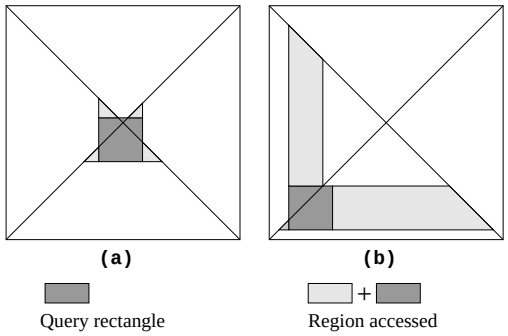


Figure 2. (a) Query rectangle near the center (b) Query rectangle near the corner

Figure 2(a) and (b) show the region accessed when the query rectangle is located near the center and corner of the data space, respectively (the dark region is the query rectangle; the lighter shaded region plus the dark region is the region accessed by the query rectangle). For uniform data, the area (or volume) of the region accessed is proportional to the number of data accessed. When the query rectangle is near the space center, most data accessed is in the answer set, so the index is efficient. However, when the query rectangle is near the space corner or edge, the data accessed is many times those in the answer set, which makes the index

quite inefficient. In other words, the effectiveness of the Pyramid technique is sensitive to the positions of the query rectangle. The space center is a "good position", while the space corner and edge are "bad positions". The difference in the access cost of these two cases is even larger in high-dimensional space. A query at a bad position will cause significant portion of the data to be accessed. The response time for processing such a query using the Pyramid technique may be longer than that of a sequential scan since sequential scan is much faster than random access of the data pages.

The performance of the Pyramid technique is also dependent on the distribution of the data set. If the data is clustered near the space corner, most queries would also be there since the distribution of queries often follows the distribution of data. This is the reason why the Pyramid technique may be worse than sequential scan for clustered data.

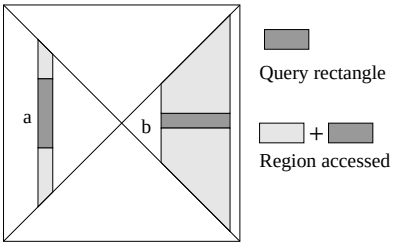


Figure 3. Non-hypercube-shaped query

Finally, the Pyramid technique is not efficient for non-hypercube-shaped queries. Figure 3 shows the region accessed by non-hypercube-shaped queries. If the query rectangle is in position a, the cost is still acceptable. But if the query rectangle is in position b, the area of the region accessed is many times the area of the query rectangle. This also makes the Pyramid technique inefficient.

Our proposed P$^+$-tree attempts to address these deficiencies. Moreover, due to its highly optimized space division and data transformation strategies, the P$^+$-tree is also efficient for kNN queries.

4 The P$^+$-tree

The basic idea of the P$^+$-tree is to divide the space into subspaces and then apply the Pyramid technique in each subspace. To realize this, we first divide the space into clusters which are essentially hyperrectangles. We then transform each subspace into a hypercube so that we can apply the Pyramid technique on it. At the same time, the transformation makes the top of the pyramids located at the cluster center. Assuming that real queries follow the same distribution as data, most of the queries would be located around the top of the pyramids, that is, the "good position". Even if some queries may be located at the corner or edge of the cluster and therefore causes a large region to be accessed, the data points accessed are not prohibitively large because most of the data points are gathered at the cluster center. In addition, the region accessed by a query is significantly reduced by space division. Thus, the P$^+$-tree can alleviate the inefficiencies of the Pyramid technique.

We note that although we cluster the space into subspaces, our scheme also works for uniform data since uniform data is a special case of clustered data. While uniform data does not benefit from the transformation, dividing the space into subspaces is still an effective mechanism for performance improvement.

To facilitate building the P$^+$-tree and query processing, we need an auxiliary structure called the *space-tree*, which is built during the space division process. The leaf nodes of the space-tree store information about the transformation. We will first introduce the data transformation, so that readers know what information is stored. Then, we present the space division process. At last, we show how the P$^+$-tree is constructed.

4.1 Data Transformation

Our transformation is motivated by the extended Pyramid technique [7], but ours is more general. As mentioned above, we have two goals: 1) transform a subspace into a unit hypercube, so that the Pyramid technique can be applied; 2) move the cluster center to the top of the pyramids, that is, the center of the unit hypercube.

A subspace is a hyperrectangle. Formally, a subspace S in a d-dimensional space is a d-dimensional interval:

$$[s_{0_{min}}, s_{0_{max}}], [s_{1_{min}}, s_{1_{max}}], ..., [s_{d-1_{min}}, s_{d-1_{max}}].$$

A window query Q is also a d-dimensional interval:

$$[q_{0_{min}}, q_{0_{max}}], [q_{1_{min}}, q_{1_{max}}], ..., [q_{d-1_{min}}, q_{d-1_{max}}].$$

Let $(c_0, c_1, ..., c_{d-1})$ be the cluster center of the subspace. Let T be a transformation on a subspace S. T consists of d functions, $t_0, t_1, ..., t_{d-1}$. Each t_i is a function on dimension i of S. To achieve the two goals mentioned in the last paragraph, T should satisfy the following conditions:

CD1. t_i is a bijection from $[s_{i_{min}}, s_{i_{max}}]$ to [0,1], that is, domain of t_i is $[s_{i_{min}}, s_{i_{max}}]$; range of t_i is [0,1]; and t_i is a one-to-one mapping.

CD2. $\forall x_1, x_2 \in [s_{i_{min}}, s_{i_{max}}]$, if $x_1 < x_2$, then $t_i(x_1) < t_i(x_2)$.

CD3. $t_i(c_i) = 0.5$

Theorem 1 *Let Q be a window query on subspace S, and T be a transformation on S that satisfies CD1 and CD2. Let $v(v_0, v_1, ..., v_{d-1})$ be a point in S. If $v \in Q$, then $T(v) \in T(Q)$ and vice versa.*

Proof See Appendix A.

Theorem 1 essentially says that the answer, say A, to a window query Q in the original space S can be obtained by operating in the transformed space $T(S)$. In other words, let the answer of the transformed query $T(Q)$ in $T(S)$ be A'. Then, $A = T^{-1}(A')$.

In fact, we do not need to find the data points in A'. We only need to identify which points in the original space that points in A' correspond to. So we store points in the original data space with the pyramid values of the transformed points as keys in the leaf nodes of a B^+-tree. This is basically the P^+-tree.

Now we need to find a transformation T that satisfies CD1, CD2 and CD3. We construct the function set as follows:

$$t_i(x) = (a_i \cdot x - b_i)^{e_i} \qquad 0 \leq i < d \qquad (1)$$

From CD1 and CD2, we can easily derive that

$$t_i(s_{i_{min}}) = 0$$

$$t_i(s_{i_{max}}) = 1$$

Plus CD3,

$$t_i(c_i) = 0.5$$

By solving the above three equations we will obtain:

$$a_i = \frac{1}{s_{i_{max}} - s_{i_{min}}} \qquad (2)$$

$$b_i = \frac{s_{i_{min}}}{s_{i_{max}} - s_{i_{min}}} \qquad (3)$$

$$e_i = -\frac{1}{\log_2(a_i \cdot c_i - b_i)} \qquad (4)$$

As we shall see shortly, in our space division scheme, we will guarantee that $s_{i_{max}} > c_i > s_{i_{min}}$, so none of the divisors is 0. In order to process the query on the transformed space $T(S)$, we need to record the transformation functions, that is, a_i, b_i, e_i. This information is stored in the leaf nodes of the space-tree.

Note that any functions that satify CD1, CD2 and CD3 can be used as T.

4.2 Space Division

As mentioned, we divide the space into clusters, but we want to keep the shape of the subspace as hyperrectangle. This is because it would be computationally expensive to determine whether a window query (which is also a hyperrectangle) intersects a subspace if the subspace is a sphere or other polygon. To achieve this, we first use a clustering method to divide all the data into two clusters. Second, we divide the space into two subspaces along the dimension in which the two cluster centers differ greatest. We apply the above two steps to subspaces recursively. Moreover, for simplicity, whenever we split a subspace further, we divide all the subspaces. So the number of subspaces is always an integral power of 2. The *Order of division* is defined as the times that we divide the space and we use Od to denote it. That is, the number of subspaces is 2^{Od}. Od is a user-defined parameter when dividing the space. We study the effects of Od in Section 6.

Since a subspace is divided into two in each division operation, we can use a binary tree, which we call the *space-tree*, to record the information of the division process. And later, we can employ it to determine whether a subspace and a query intersects.

The space-tree is similar to the k-d tree [5], but we store the transformation information instead of data points in the leaf nodes. In the space-tree, a nonleaf node contains 4 items: DD, DV, PL, PR. DD is an integer denoting the dimension in which the space is divided. DV is a real number denoting the value the space is divided. PL and PR are pointers pointing to the two subspaces. A leaf node of the space-tree also contains 4 items: SNo, $a[d]$, $b[d]$, $e[d]$. SNo is an integer ranging from 0 to $2^{Od} - 1$, which identifies a subspace. SNo is assigned to each subspace when dividing the space. $a[d], b[d]$ and $e[d]$ are three arrays which store the transformation information of subspace SNo; they are calculated according to Equations (2), (3) and (4). Figure 4 shows a 2-dimensional example. Here, $Od = 2$. Figure 5 is the space-tree for the space division in Figure 4.

The space-tree file is very small, typically ten to several hundred KiloBytes. Thus, it can be kept in memory to accelerate searching.

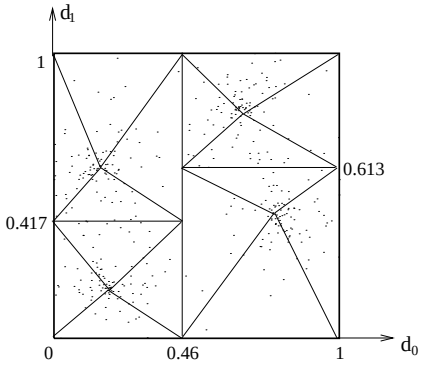


Figure 4. Space division and data transformation

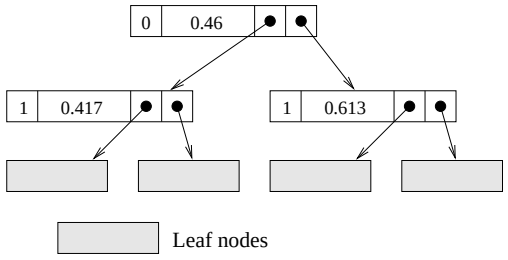


Figure 5. Space-tree

The space division algorithm is outlined below.

Algorithm **Space Division**
SD1 $space\ 0$ = the original data space.
SD2 for ($cdt = 0; cdt < Od; cdt++$)
SD3 for ($n = 0; n < 2^{cdt}; n++$)
SD4 Divide all data in $space\ n$ into two clusters, the cluster centers are CL and CR
SD5 The dividing dimension $DD = m$, where $|CL_m - CR_m| = Max(|CL_j - CR_j|, 0 \leq j < d)$.
SD6 Dividing value $DV = \frac{CL_{DD}+CR_{DD}}{2}$.
SD7 Divide $space\ n$ into two subspaces SL and SR according to DD and DV, that is, for each $v \in$ space n, if $v_{DD} < DV$, then $v \in SL$; else $v \in SR$.
SD8 Assign space numbers to the subspaces. SNo of SL is $2n$; SNo of SR is $2n + 1$.
SD9 Record $S_{i_{min}}$ and $S_{i_{max}}$ of SL and SR.
SD10 Insert a nonleaf node to space-tree.
//Now we have divided the space Od times and have
//2^{Od} subspaces with SNo from 0 to $2^{Od} - 1$.
SD11 for ($n = 0; n < 2^{Od}; n++$)
SD12 Calculate center of all points in space n, so we get c_i, $0 \leq i < d$.
SD13 Calculate a_i,b_i,e_i according to Equation (2) (3) and (4), $0 \leq i < d$.
SD14 Insert a leaf node to space-tree.
End Space Division

In the algorithm, cdt means current dividing time. There are a plethora of clustering algorithms, any one can be used in SD4. In our study, we used the Bisecting K-means algorithm as described in [18]. In SD8, we use a flag to make the SNos of the subspaces effective after cdt increases so that they would not affect other subspaces in the current round of space division.

4.3 Construction of the P$^+$-tree

A P$^+$-tree is basically a B$^+$-tree where the data records with their keys are stored in the leaf nodes of the P$^+$-tree. In the P$^+$-tree, we apply the Pyramid technique in each subspace. Under the Pyramid technique, pyramid values of points in pyramid i cover the interval $[i, i + 0.5]$. There are $2d$ pyramids from pyramid 0 to pyramid $2d-1$, so pyramid values of all points are within the interval $[0, 2d)$. To discriminate points from different subspaces, we add $SNo \cdot 2d$ to the pyramid value of a point in subspace SNo and the result is the key for the point.

Algorithm **Build P$^+$-tree**
BP1 for ($n = 0; n < 2^{Od}; n++$)
BP2 Read the leaf node for space n from space-tree
BP3 for each point v in space n
BP4 for ($i = 0; i < d; i++$)
BP5 $v'_i = (a_i \cdot v_i - b_i)^{e_i}$
BP6 key=$n \cdot 2d$+***pv***(v')
BP7 BtreeInsert(v, key);
End Build P$^+$-tree

In the algorithm, ***pv***( ) is the function to calculate pyramid value as defined in Section 3.

The above algorithm builds a P$^+$-tree after the space has been split, so we know which subspace a point belongs to. If we insert a point or delete a point after the P$^+$-tree is built, we need to traverse the space tree to decide which subspace a point belongs to by the following algorithm.

Algorithm **DetermineSubspace**($node$)
DS1 if $node$ is nonleaf node
DS2 Read DD,DV,PL,PR from $node$.
DS3 if $q_{DD} < DV$ ***DetermineSubspace***(PL);
DS4 if $q_{DD} \geq DV$ ***DetermineSubspace***(PR);
DS5 if $node$ is leaf node
DS6 return $node$
End DetermineSubspace

The leaf node returned by the DetermineSubspace algorithm contains the SNo, a_i, b_i, e_i of the subspace. Then we can calculate the key of the point $v(v_0, v_1, ..., v_{d-1})$.

$$key = SNo \cdot 2d + \textbf{\textit{pv}}(v')$$

where $v'_i = (a_i \cdot v_i - b_i)^{e_i}, \quad 0 \leq i < d$

5 Query Processing

In this section, we shall look at how the P$^+$-tree can be used to support window and kNN queries.

5.1 Window Queries

In the P$^+$-tree, window queries are processed in two logical phases. Let Q be a window query. In the first phase, we determine the clusters that are intersected by Q. In this way, other clusters can be pruned. This can be easily done by traversing the space-tree. As the space-tree is traversed from the root, we check which cluster is intersected by Q based on the split dimension and split value, and if one is intersected, we examine its child nodes recursively.

The second phase operates on an intersected subspace S. First, we need to transform the query to $T(Q)$. Then we process query $T(Q)$ on the transformed subspace $T(S)$ using the Pyramid technique. Specifically, the portion of $T(Q)$ within each subspace is mapped to intervals of the form $[h_{low}, h_{high}]$; if pyramid i is intersected by the query, a B$^+$-tree range query function is invoked to find those candidate

points for $T(Q)$. Because these candidate points are the transformed points, they correspond to the original points which are candidates for Q according to Theorem 1. So we only need to check whether the original candidate points are within Q to get our final answers.

We note that given a subspace S, some part of Q may be outside of S. In this case, $T(Q)$ is outside of the unit hypercube accordingly. We can just cut off the range of $T(Q)$ that is greater than 1 or less than 0 and then we get the part within the unit hypercube, which corresponds to the part of Q within S. The window search algorithm is presented below.

Algorithm **Window Search**
WS1: ***TraverseSpaceTreeWindow***(space-tree root);
End Window Search

Algorithm **TraverseSpaceTreeWindow**($node$)
TW1 if $node$ is nonleaf node
TW2 Read DD,DV,PL,PR from $node$.
TW3 if $q_{DD} < DV$ ***TraverseSpaceTreeWindow***(PL);
TW4 if $q_{DD} \geq DV$ ***TraverseSpaceTreeWindow***(PR);
TW5 if $node$ is leaf node
TW6 Read SNo and transformation information T
TW7 $Q' = T(Q)$;
TW8 Cut off the part of Q' that is outside unit hypercube
TW9 for each pyramid in space SNo
TW10 if ***intersect***(p_i, Q')
TW11 ***determin_range***(p_i, Q',h_{low}, h_{high})
TW12 $candidates$=***BtreeRangeSearch***(
$SNo \cdot 2d + i + h_{low}$,
$SNo \cdot 2d + i + h_{high}$)
TW13 for each point v in $candidates$
TW14 Check if v is within Q, if yes, add it to the answer set.
End TraverseSpaceTreeWindow

In the algorithm, ***intersect***() is the function to determine whether a pyramid is intersected by a query rectangle. As we mentioned in Section 3, a query rectangle corresponds to a height range in an intersected pyramid. ***determin_range***() is the function to determine the height range $[h_{low}, h_{high}]$ according to the pyramid and query. ***BtreeRangeSearch***() is a standard B$^+$-tree range search function to retrieve all the records with the keys in the given range.

5.2 KNN Queries

To find the kNN of a point x, we initiate a hypercube-shaped window query centered at x with an initial side length, which is typically small. Then we increase the side length gradually until we are sure that the kNNs are found. Here we also traverse the space-tree, but the processing of each intersected subspace is different from that of an window query. When we enlarge the query rectangle, the center part of it has been searched the last time. In order to avoid searching it again, we use three arrays, $flag[], lp[], rp[]$, to record whether a pyramid has been searched; and if so, the leaf nodes we should start from.

Algorithm **KNN Search**
KS1 $sl = sl_0,\ A = \emptyset$; /*initialize side length of the query and answer set*/
KS2 initialize $flag[], lp[], rp[]$;
KS3 $v_{farthest} =$ ***farthest***(A, x);
KS4 while ***distance***$(v_{farthest}, x) > sl/2$ or $|A| < k$;
KS5 $sl = sl + dl$;
KS6 ***TraverseSpaceTreeKNN***(space-tree root);
End KNN Search

Algorithm **TraverseSpaceTreeKNN**($node$)
TK1 if $node$ is nonleaf node
TK2 Read DD,DV,PL,PR from $node$.
TK3 if $q_{DD} < DV$ ***TraverseSpaceTreeKNN***(PL);
TK4 if $q_{DD} \geq DV$ ***TraverseSpaceTreeKNN***(PR);
TK5 if $node$ is leaf node
TK6 Read SNo and transformation information T from $node$
TK7 $Q' = T(Q)$;
TK8 Cut off the part of Q' that is outside unit hypercube.
TK9 for each pyramid in space SNo
TK10 if ***intersect***(p_i, Q')
TK11 ***determin_range***(p_i, Q',h_{low}, h_{high})
TK12 if not $flag[SNo \cdot 2d + i]$ /*If this pyramid has not been searched before*/
TK13 $lnode =$***LocateLeaf***$(SNo \cdot 2d + i + h_{low})$
TK14 $lp[SNo \cdot 2d + i] = lnode$
TK15 $rp[SNo \cdot 2d + i] =$***SearchUp***(
$lnode, SNo \cdot 2d + i + h_{high}$)
TK16 $flag[SNo \cdot 2d + i] =$TRUE
TK17 else
TK18 if $lp[SNo \cdot 2d + i]$ not NULL
TK19 $lp[SNo \cdot 2d + i] =$***SearchDown***(
$lp[SNo \cdot 2d + i] \rightarrow leftnode$,
$SNo \cdot 2d + i + h_{low}$)
TK20 if $rp[SNo \cdot 2d + i]$ not NULL
TK21 $rp[SNo \cdot 2d + i] =$***SearchUp***(
$rp[SNo \cdot 2d + i] \rightarrow rightnode$,
$SNo \cdot 2d + i + h_{high}$)
End TraverseSpaceTreeKNN

Algorithm **SearchUp**($node, limit$)
SU1 for each point v in $node$
SU2 if $|A| == k$
SU3 $v_{farthest} =$ ***farthest***(A, x)
SU4 if ***distance***$(v, x) <$***distance***$(v_{farthest}, x)$

SU5 $A = A - v_{farthest}$
SU6 $A = A \cup v$
SU7 else
SU8 $A = A \cup v$
SU9 if key of the last point in $node < limit$
SU10 $node =$***SearchUp***$(node \rightarrow rightnode, limit)$
SU11 if end of this pyramid is reached
SU12 $node$=NULL
SU13 else
$node$=NULL
SU14 return $node$
End SearchUp

Among all the points in A, ***farthest***(A, x) returns the farthest one to x. ***distance***() returns the distance between two points. ***LocateLeaf***(key) returns the address of the leaf page which contains the key. ***SearchUp***$(lnode, limit)$ starts searching from $lnode$ and search its right sibling leaf node recursively until the key in the node reaches the $limit$. At the same time, the last node accessed is returned and stored in $rp[]$. ***SearchDown***() is similar to ***SearchUp***(), so we only present the algorithm of ***SearchUp***().

6 Performance Study

In this section, we present results from an extensive performance study to evaluate the P$^+$-tree for window queries and kNN queries under a wide range of workloads. We have generated synthetic clustered and uniform data sets of different sizes varying from 100,000 to 2,000,000 points and of different dimensionality. We also used two real data sets. Besides, we varied the selectivity and shape of queries. For each experiment setting, we run 200 queries and use the average number of page accesses or average total response time as the performance metric. The distribution of the queries are the same as the data. All experiments are run on a computer with Pentium(R) 1.6GHz CPU and 256MB RAM. The page size is 4096 Bytes.

6.1 Window queries

For window queries, the Pyramid technique has been shown to outperform the X-tree and the Hilbert-R-tree [7] and the iMinMax has been shown to be superior over the Pyramid technique for skewed data [15]. Thus, as references, we compare the P$^+$-tree with these two methods and sequential scan.

6.1.1 Synthetic Clustered Data

In this set of experiments, we study the P$^+$-tree for window queries on clustered data. We have done experiments on data sets with different dimensionality and sizes. As default, we use the data set with 24-dimensions and 1,000,000 data points. The data we use have 4 natural clusters. Figure 6 shows a 2-dimensional image of the data distribution. For every series of experiments, the iMinMax is tuned and the optimal θ is used.

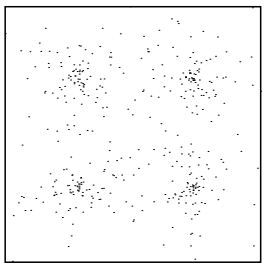


Figure 6. Data distribution of clustered data

Effects of Order of Division In the first experiment, we would like to tune the P$^+$-tree to determine the optimal order to use for division. Intuitively, the larger the number of subspaces, the smaller will the subspaces be. On the one hand, this means that we can prune a larger portion of the data space that does not intersect the query. On the other hand, the overhead of accessing the subspaces increases – for each accessed subspace, we need to access at least 1 leaf node, but each page may only contain a few points in the answer set.

Figure 7 shows the effects of different Od as we vary the selectivities of queries. We note that a large Od is superior over a small Od at low selectivities but a small Od outperforms a large Od at high selectivities. While there is an optimal Od for different selectivities, the relative difference between them is not big, so a moderate one is fine.

In view of these results, we used 6 as the default Od of the P$^+$-tree.

Effects of Selectivity Figures 8 and 9 show the performance comparison for queries of different selectivities. The number of page accesses of the P$^+$-tree is 20% to 40% that of the Pyramid Technique. The iMinMax performs as well as the Pyramid technique when the selectivity is low, but better than the Pyramid technique for high selectivity. In terms of total response time, the Pyramid technique and the iMinMax are better than sequential scan at low selectivities but worse at high selectivities. The P$^+$-tree is always the best and achieves a speedup factor of 2.4 to 2.8 over the Pyramid Technique and iMinMax, and 1.7 to 4.9 over sequential scan. It is reasonable that as selectivity increases, the number of page accesses becomes larger. When the number of page accesses of these indexing schemes increase to some point, the total response time exceeds that of the sequential scan, because sequential access of the disk is much faster than random accesses using the indexing structures. In the P$^+$-tree, the space division effectively prune the subspace not intersected by the query rectangle and false positives in the intersected subspaces are greatly reduced by

the data transformation. Therefore, the number of page accesses are kept at a very low level even in the case of a high selectivity.

Due to space limitations, for all subsequent experiments, we shall only present the total response time results since it reflects more accurately the relative performance of the various schemes. However, we note that the relative performance of the schemes in term of the number of page accesses are largely the same.

Effects of Data set size In these experiments, we vary the data set sizes from 100,000 to 2,000,000 points. As shown in Figure 10, the Pyramid Technique and the iMinMax outperform sequential scan for small data sets. However, as the data set size increases, their performance deteriorate quickly to the extent that they were outperformed by sequential scan. The P^+-tree has a speedup factor of 3.8 to 5.1 over the Pyramid Technique and the iMinMax, and 3.7 to 6.1 over sequential scan. The results show that the P^+-tree scales well with data set sizes.

Effects of Dimensionality To see the effects of dimensionality on the P^+-tree, we also experimented with 8, 16, 24, 32, 64 and 128 dimensional data. In these experiments, we fixed the selectivities of the queries to 2%. The results in Figure 12 clearly show that the P^+-tree does not suffer from the curse of dimensionality. It has a speedup factor of 3.2 over the Pyramid Technique and iMinMax, and always outperforms sequential scan.

Effects of Number of clusters All the above experiments are tested on data that have 4 natural clusters. To see how the number of clusters affects the performance of the techniques, we also did experiments on data having 2 and 3 natural clusters, which follow the distribution as shown in Figure 11. Figures 13 and 14 show the results.

The P^+-tree still performs best, but we found that the performance improvement increases as the number of clusters increases. Comparing the speedup factor of the P^+-tree over the Pyramid Technique for the queries of selectivity 2%, the speedup factor for 2-, 3- and 4- cluster data is 2.2, 2.7 and 2.8 respectively. On the other hand, the iMinMax performs better compared to the Pyramid Technique as the number of clusters decreases. This is because θ in the iMinMax is a global parameter, and the tuning of θ would be more effective for data with fewer clusters and are very skewed. As shown, even after tuning, the P^+-tree outperforms the iMinMax by a wide margin.

6.1.2 Synthetic Uniform Data

Figure 15 shows the performance comparison for queries with the selectivity of about 0.05% on data of different dimensionality. As shown, the P^+-tree is slightly better than the Pyramid technique and the iMinMax, with a smaller speedup factor over them than in the clustered data case.

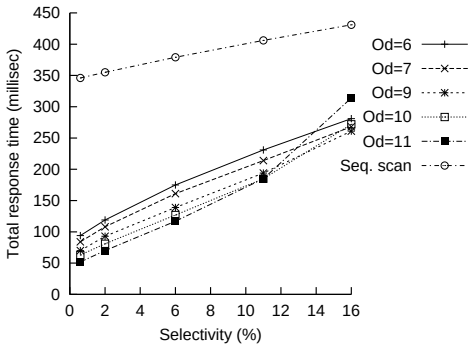


Figure 7. Effects of Order of Division

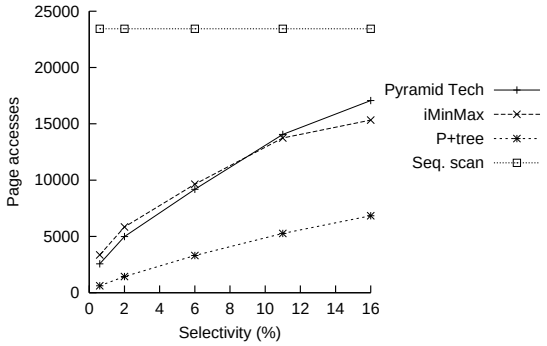


Figure 8. Page accesses vs. Selectivity

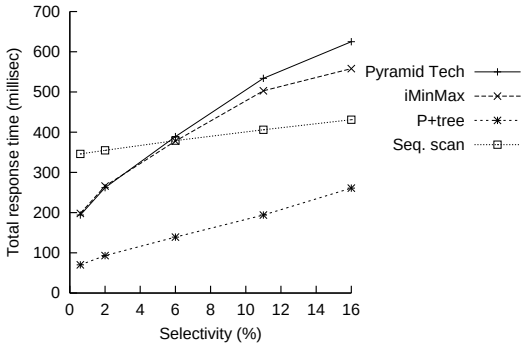


Figure 9. Total response time vs. Selectivity

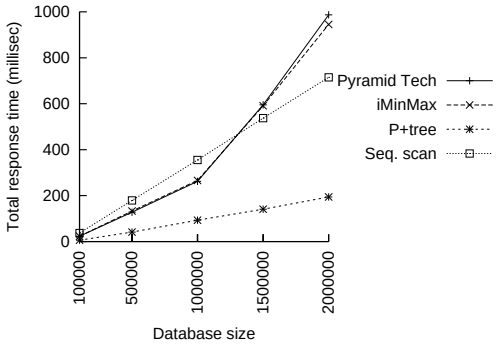
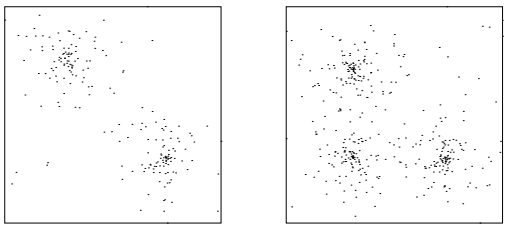


Figure 10. Effects of Data set size



(a) 2 Clusters (b) 3 Clusters

Figure 11. Data distribution of different number of clusters

This is because the Pyramid technique is already good for uniform data.

Figure 16 shows the performance comparison over varying query side length. The speedup factor of the P^+-tree over the Pyramid technique and the iMinMax is large at very small or very large side lengths while they are very close at medium side length. This is because the Pyramid technique is primarily designed and optimized for queries of medium side length on uniform data, but it does not work well with very small or very big sized queries. The P^+-tree overcomes the deficiencies and performs better than sequential scan even for very large query rectangles.

6.1.3 Partial Window Queries

All the queries tested so far are hypercube shaped. Now we test non-hypercube-shaped queries. Partial window queries are the worst case of non-hypercube-shaped queries but frequently used in real applications. So we use them in our experiments as representatives of non-hypercube-shaped queries. In this experiment we use 24-dimensional data and only set range limits in 6 dimensions, that is, the other 18 dimensions of the queries are full domains. Figures 17 and 18 show the performance on clustered data and uniform data respectively.

For clustered data, the P^+-tree has a speedup factor of 3 over the Pyramid technique and 2 to 4.6 over sequential scan. The iMinMax is better than the Pyramid technique when selectivity is low but deteriorate rapidly and outperformed by the other techniques at a higher selectivity. The Pyramid technique is always worse than sequential scan.

For uniform data, the total response time of the Pyramid technique and the iMinMax is far more than that of sequential scan. The P^+-tree has a speedup factor of 2.4 to 4.5 over the Pyramid technique and the iMinMax. The P^+-tree also outperforms the sequential scan. Its performance approaches sequential scan as selectivity increases.

The Pyramid technique and the iMinMax are inefficient for partial window queries because by their indexing schemes, they can hardly prune data if there are full domains in the winodw query. But the P^+-tree can still prune subspaces because of the space division.

6.1.4 Real Data

We have tested the techniques on the following two real data sets:

1) Forest CoverType (10 dimensions)
The forest cover type data set for 30 x 30 meter cells is obtained from the US Forest Service (USFS) Region 2 Resource Information System (RIS) data. The original data set has 54 attributes, among which 10 are quantitative such as *Elevation*, *Aspect* and *Slope*. We extracted these 10 quantitative attributes and normalized the values to the range [0,1]. The number of records is 581,012. The original data set is available online at [1].

2) Color Histogram (32 dimensions)
This data set contains image features extracted from a Corel image collection. HSV color space is divided into 32 subspaces (32 colors: 8 ranges of H and 4 ranges of S). And the value in each dimension in a ColorHistogram of an image is the density of each color in the entire image. The number of records is 68,040. This data set is available online at [2].

These two data sets are of medium- and high- dimensionality respectively. We set Od as 3 for these two data sets because of their smaller sizes. Figures 19 and 20 show the comparison on total response time. Horizontal axes of the figures represent the number of attributes specified in the partial window queries. The larger the number of attributes specified in the window queries, the fewer the data points in the answer set and therefore the shorter the time needed for processing the queries. In most cases, the P^+-tree has a speedup factor of 2 to 5 over the other techniques. These figures further confirm the practical impact of the P^+-tree.

6.2 KNN Queries

Experiments in [17] show that the A-tree outperforms the SR-tree and the VA-file, while [21] shows that the iDistance outperforms the A-tree. So we compared the P^+-tree with the iDistance. Moreover, for kNN queries, sequential scan is shown to be more efficient for uniform data [19]. Thus, we shall only consider synthetic clustered data and real data. Two synthetic data sets are used: 16-dimensional and 32-dimensional, data set size 500,000, 4 natural clusters. The real data set is the 32-dimensional Color Histogram data set.

Figure 21 shows the result for 16-dimensional clustered data set. The P^+-tree has a speedup factor of up to 1.6 over the iDistance and both of them are much better than sequential scan. Although kNN search of the P^+-tree is based on window search, it benefits from the highly efficient space division and data transformation strategies. The iDistance uses hypersphere-like clusters. A problem is that in high-dimensional space, the radius of each cluster becomes very

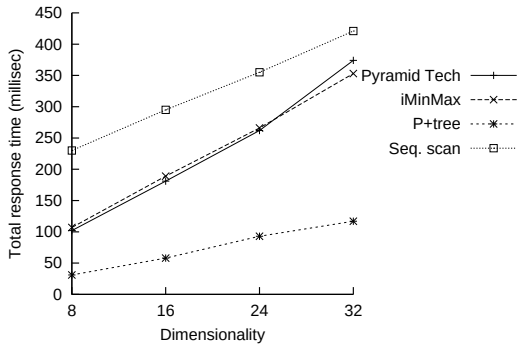


Figure 12. Effects of Dimensionality

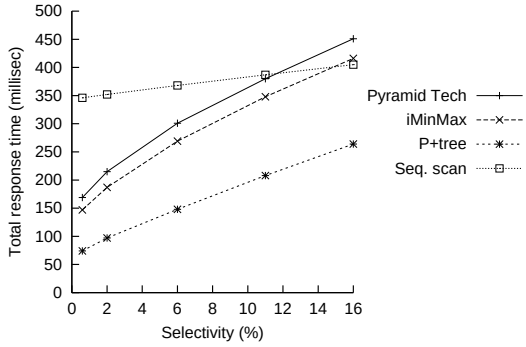


Figure 13. Effects of Number of clusters, 2 clusters

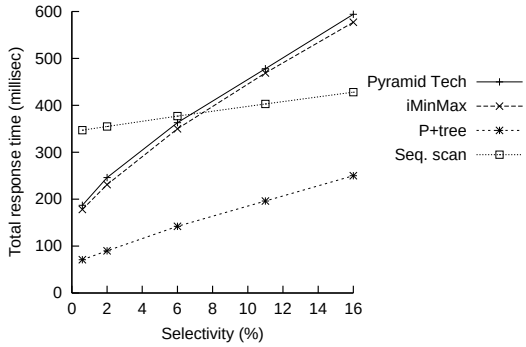


Figure 14. Effects of Number of clusters, 3 clusters

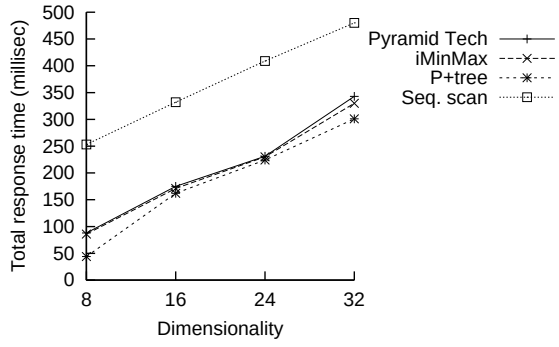


Figure 15. Effects of Dimensionality, Uniform data

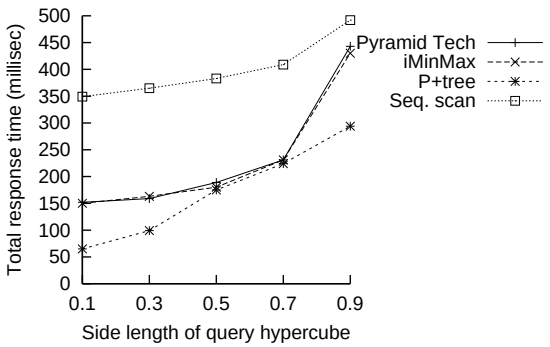


Figure 16. Total response time vs. Side length of query hypercube, Uniform data

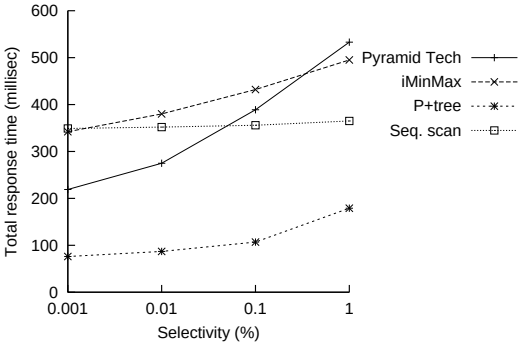


Figure 17. Partial window queries on clustered data

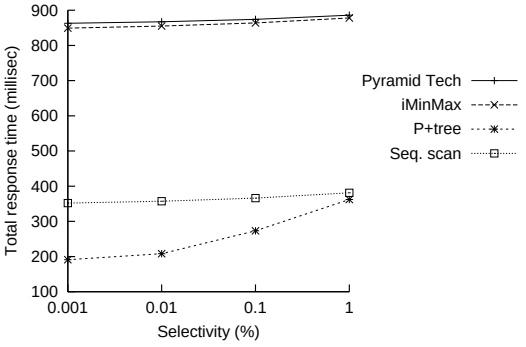


Figure 18. Partial window queries on uniform data

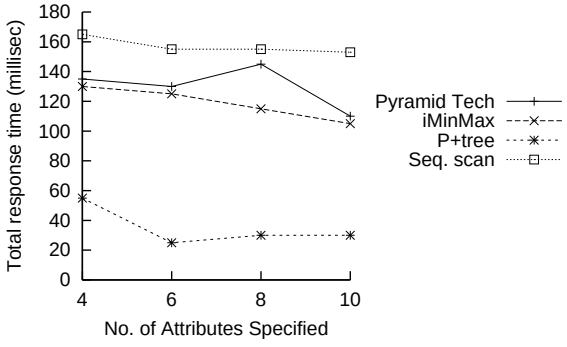


Figure 19. Forest CoverType data, Window queries

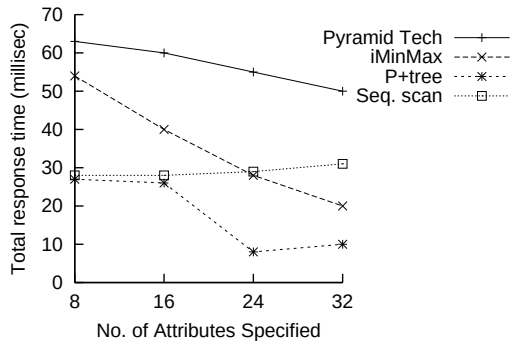


Figure 20. Color Histogram data, Window queries

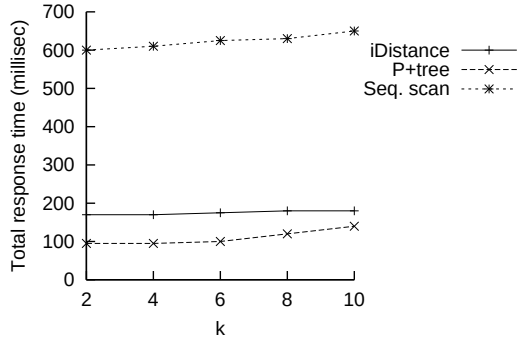


Figure 21. Synthetic 16-dimensional clustered data, kNN queries

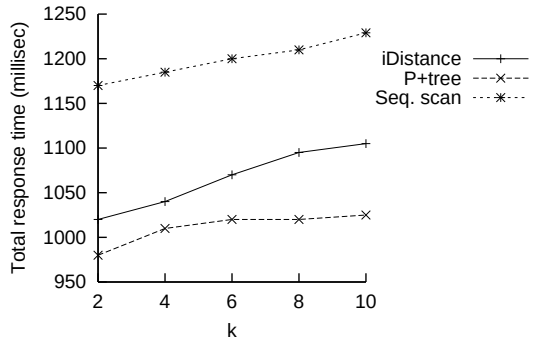


Figure 22. Synthetic 32-dimensional clustered data, kNN queries

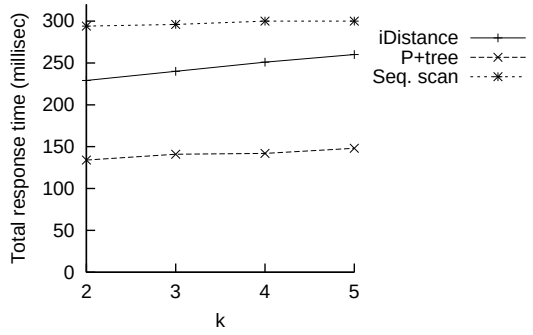


Figure 23. Color Histogram data, kNN queries

large. This results in almost every cluster being intersected by a kNN query sphere, so the effect of pruning may not be good. In this case, a simple division in the space may prune unnecessary access more effectively. Figures 22 and 23 show the results for 32-dimensional clustered data set and the color histogram data set. The results show similar trend, but the speedup of the iDistance and the P^+-tree over sequential scan is smaller. The reason is that for higher dimensionality, indexing schemes generally become less efficient compared to their performance in lower dimensionality. In the experiments shown in Figure 23, we used smaller values of k because the color histogram data set is relatively small.

6.3 On Updates

The space division process is based on clustering the entire data set. If new data points are inserted or deleted from the data set, the cluster center of the subspace may shift from the top of the pyramids. While this does not affect the correctness of the P^+-tree, it may affect its efficiency. To evaluate how updates affect the performance of the P^+-tree, we have done the following experiments. We use a synthetic 24-dimensional clustered data set with 500,000 points. We first construct the P^+-tree using 80% (400,000) of the data. We run some window queries and record the average total response time. Then we insert 5% of the data to the database and re-run the same queries. This process is repeated until the other 20% of the data are inserted. We keep the query constant. On the other hand, we run the queries on the P^+-tree built when there are 85%, 90%, ... of data. That is, the optimal P^+-tree with no updates. We compare the total response time of the two as shown in Figure 24. As expected, the difference between them becomes larger as more data are inserted, but the largest difference is within 20% even in the case of 20% newly inserted data. Considering that the P^+-tree typically outperforms other techniques by the factor of 2 to 4, this deterioration is acceptable. Experiments on kNN queries have similar results.

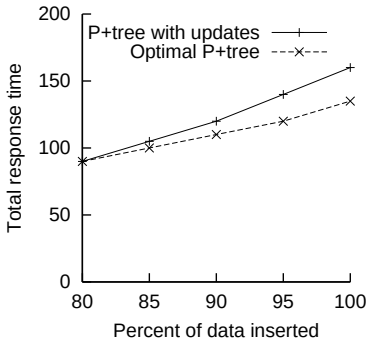


Figure 24. P^+-tree performance with updates

Of course, we expect the performance of the P^+-tree to be affected greatly eventually if there are too many updates. In this case, we need to rebuild the P^+-tree. Fortunately, rebuilding the P^+-tree is not quite expensive since bulk loading of B^+-tree still applies. Note that any clustering method can be used in SD4 of the Space Division Algorithm. This means that we can use approximate but fast methods to obtain suboptimal clusters to reduce the time for space division.

An interesting phenomenon is that building a P^+-tree is much faster than building a Pyramid-tree or an "iDistance-tree". The reason is that, space division is in effect a clustering process. Moreover, in the P^+-tree, we add a number $SNo \cdot 2d$ to the pyramid values of points in the subspace. So keys for points from different subspaces are scattered. Data points in the same subspace have close keys and are inserted into the tree continuously. A single page I/O is needed for many continuous insertions. Therefore, many disk accesses are avoided. In our experiments, the time saved in building the P^+-tree offsets most of the time used in space division.

7. Conclusion

In this paper, we have proposed the P^+-tree for processing multi-dimensional queries. First, we divide the space into subspaces that are subsequently transformed so that the Pyramid technique can be applied in each subspace. The transformation function also moves the center of the points in a subspace to the top of pyramids. A major strength of the P^+-tree is that it works well for various workloads. Extensive experiments demonstrate that the P^+-tree has considerable speedup over existing indexing methods for both window queries and kNN queries.

APPENDIX

A. Theorem 1

Let Q be a window query on subspace S, and T be a transformation on S that satisfies CD1 and CD2. Let $v(v_0, v_1, ..., v_{d-1})$ be a point in S. If $v \in Q$, then $T(v) \in T(Q)$ and vice versa.

Proof If $v \in Q$, then

$$q_{i_{min}} < v_i < q_{i_{max}}, \qquad 0 \le i < d$$

According to CD2,

$$t_i(q_{i_{min}}) < t_i(v_i) < t_i(q_{i_{max}}), \qquad 0 \le i < d$$

that is, $T(v) \in T(Q)$.

If $T(v) \in T(Q)$, then

$$t_i(q_{i_{min}}) < t_i(v_i) < t_i(q_{i_{max}}), \qquad 0 \le i < d$$

According to CD1 and CD2, T is a strictly increasing bijection, therefore

$$q_{i_{min}} < v_i < q_{i_{max}}, \qquad 0 \le i < d$$

that is, $v \in Q$. □

References

[1] *http://kdd.ics.uci.edu/databases/covertype/covertype.html*.

[2] *http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html*.

[3] M. Arya, W. Cody, C. Faloutsos, J. Richardson, and A. Toga. Qbism: Extending a dbms to support 3d medical images. In *ICDE*, 1994.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, 1975.

[6] S. Berchtold, C. Bohm, H. V. Jagadish, and J. S. Hans-Peter Kriegel. Independent quantization: An index compression technique for high-dimensional data spaces. In *ICDE*, 2000.

[7] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *SIGMOD*, 1998.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, 1997.

[9] A. P. de Vries, N. Mamoulis, N. Nes, and M. Kersten. Efficient knn search on vertically decomposed data. In *SIGMOD*, 2002.

[10] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 1994.

[11] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[12] H. V. Jagadish. A retrieval technique for similar shapes. In *SIGMOD*, 1991.

[13] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB*, 1994.

[14] N. Katayama and S. Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *SIGMOD*, 1997.

[15] 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 *PODS*, 2000.

[16] H.-P. K. S. Berchtold, D. Keim. The x-tree: An index structure for high-dimensional data. In *VLDB*, 1996.

[17] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The a-tree: an index structure for high-dimensional spaces using relative approximation. In *VLDB*, 2000.

[18] Savaresi, S.M., Boley, D.L., Bittanti, S., and Gazzaniga. Cluster selection in divisive clustering algorithms. In *Proc. SIAM Int. Conf. on Data Mining*, 2002.

[19] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[20] D. A. White and R. Jain. Similarity indexing with the ss-tree. In *ICDE*, 1996.

[21] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB*, 2001.