The Pyramid-Technique: Towards Breaking the Curse of Dimensionality

Stefan Berchtold
AT&T Labs Research
Florham Park, NJ
berchtol@research.att.com

Christian Böhm
University of Munich
Germany
boehm@informatik.uni-muenchen.de

Hans-Peter Kriegel
University of Munich
Germany
kriegel@informatik.uni-muenchen.de

Abstract

In this paper, we propose the Pyramid-Technique, a new indexing method for high-dimensional data spaces. The Pyramid-Technique is highly adapted to range query processing using the maximum metric L_{max}. In contrast to all other index structures, the performance of the Pyramid-Technique does not deteriorate when processing range queries on data of higher dimensionality. The Pyramid-Technique is based on a special partitioning strategy which is optimized for high-dimensional data. The basic idea is to divide the data space first into 2d pyramids sharing the center point of the space as a top. In a second step, the single pyramids are cut into slices parallel to the basis of the pyramid. These slices form the data pages. Furthermore, we show that this partition provides a mapping from the given d-dimensional space to a 1-dimensional space. Therefore, we are able to use a B+-tree to manage the transformed data. As an analytical evaluation of our technique for hypercube range queries and uniform data distribution shows, the Pyramid-Technique clearly outperforms index structures using other partitioning strategies. To demonstrate the practical relevance of our technique, we experimentally compared the Pyramid-Technique with the X-tree, the Hilbert R-tree, and the Linear Scan. The results of our experiments using both, synthetic and real data, demonstrate that the Pyramid-Technique outperforms the X-tree and the Hilbert R-tree by a factor of up to 14 (number of page accesses) and up to 2500 (total elapsed time) for range queries.

1 Introduction

During recent years, a variety of new database applications has been developed which substantially differ from conventional database applications in many respects. For example, new database applications such as data warehousing [11] produce very large relations which require a multidimensional view on the data, and in areas such as multimedia [16, 20] a content-based search is essential which is often implemented using some kind of feature vectors. All the new applications have in common that the underlying database system has to support query processing on large amounts of high-dimensional data. Now, the reader may ask what the difference is between processing low- and high-dimensional data. A result of recent research activities [5, 6, 23] is that basically none of the querying and indexing techniques which provide good results on low-dimensional data also performs sufficiently well on high-dimensional data for larger queries. The only approach taken to solve this problem for larger queries was parallelization [2]. In this paper, however, we will tackle the problems leading to the so-called curse of dimensionality. A variety of new index structures [18, 19], cost models [5, 14] and query processing techniques [7, 4] have been proposed. Most of the index structures are extensions of multidimensional index structures adapted to the requirements of high-dimensional indexing. Thus, all these index structures are restricted with respect to the data space partitioning. Additionally, they suffer from the well-known drawbacks of multidimensional index structures such as high costs for insert and delete operations and a poor support of concurrency control and recovery.

Motivated by these disadvantages of state-of-the-art index structures for high-dimensional data spaces, we developed the Pyramid-Technique. The Pyramid-Technique is based on a special partitioning strategy which is optimized for high-dimensional data. The basic idea is to divide the data space such that the resulting partitions are shaped like peels of an onion. Such partitions cannot be efficiently stored by R-tree-like index structures. However, we achieve the partitioning by first dividing the d-dimensional space into $2d$ pyramids having the center point of the space as their top. In a second step, the single pyramids are cut into slices parallel to the basis of the pyramid forming the data pages. As we will show both analytically and experimentally, this strategy outperforms other partitioning strategies when processing range queries. Furthermore, we will analytically show that range query processing using our method is not affected by the so-called "curse of dimensionality" i.e., the performance of the Pyramid-Technique does not deteriorate when going to higher dimensions. Instead, the performance improves for increasing dimension. Note that this analytical result is obtained for hypercube shaped queries and uniform data distribution. Queries, which touch the boundary of the data space, or very skewed queries are handled less efficiently. However, as we will show in the experimental section of this paper, even slightly skewed queries can be handled efficiently.

Another advantage of the Pyramid-Technique is the fact that we use a mapping from the given d-dimensional data space to a 1-dimensional space in order to achieve the mentioned onion-like partitioning. Therefore, we can use a B^+-tree [1, 10] to store the data items and take advantage of all the nice properties of B^+-trees such as fast insert, update and delete operations, good concurrency control and recovery, easy implementation and re-usage of existing B^+-tree implementations. The Pyramid-Technique can easily be implemented on top of an existing DBMS.

The rest of this paper is organized as follows: In section 2, we give an overview of the related work in high-dimensional indexing and show how the Pyramid-Technique is related to this work. In section 3, we analyze the behavior of the space partitioning strategy traditionally used by multidimensional index structures. In section 4 and section 5 we present our new method, especially fo-

SIGMOD '98 Seattle, WA, USA

cusing on the query processing algorithm of the Pyramid-Technique. Then, we analyze in section 6 the benefits of the Pyramid-Technique. To improve the performance of the Pyramid-Technique in case of real data, we propose some extensions of the Pyramid-Technique in section 7. Finally, we present a variety of experiments demonstrating the practical impact of our technique. A discussion of the weaknesses and limitations of the Pyramid Technique will conclude the paper.

2 Related Work

Recently, a few high-dimensional index structures have been proposed.

Lin, Jagadish and Faloutsos presented the TV-tree [19] which is an R-tree-like index structure. The central concept of the TV-tree is the telescope vector (TV). Telescope vectors divide attributes into three classes: attributes which are common to all data items in a subtree, attributes which are ignored and attributes which are used for branching in the directory. The motivation for ignoring attributes is that a sufficiently high selectivity can often be achieved by considering only a subset of the attributes. Therefore, the remaining attributes have no chance to substantially contribute to query processing. Obviously, redundant storage of common attributes does not contribute to query processing either. The major drawback of the TV tree is that information about the behavior of single attributes, e.g. their selectivity, is required.

Another R-tree-like high-dimensional index structure is the SS-tree [23] which uses spheres instead of bounding boxes in the directory. Although the SS-tree clearly outperforms the R*-tree, spheres tend to overlap in high-dimensional spaces. Thus, recently a improvement of the SS-tree has been proposed in [18], where the concepts of the R-tree and SS-tree are integrated into a new index structure, the SR-tree. The directory of the SR-tree consists of spheres (SS-tree) and hyper-rectangles (R-tree) such that the area corresponding to a directory entry is the intersection between the sphere and the hyper-rectangle. Therefore, the SR-tree outperforms both the R*-tree and the SS-tree.

In [17], Jain and White introduced the VAM-Split R-tree and the VAM-Split KD-tree. Both are static index structures i.e. all data items must be available at the time of creating the index. VAM-Split trees are rather similar to KD-trees [21], however in contrast to KD-trees, splits are not performed using the 50%-quantile of the data according to the split dimension, but on the value where the maximum variance can be achieved. VAM Split trees are built in main memory and then stored on secondary storage. Therefore, the size of a VAM Split tree is limited by the main memory available during the creation of the index.

In [6], the X-tree has been proposed which is an index structure adapting the algorithms of R*-trees to high-dimensional data using two techniques: First, the X-tree introduces an overlap-free split algorithm which is based on the split history of the tree. Second, if the overlap-free split algorithm would lead to an unbalanced directory, the X-tree omits the split and the according directory node becomes a so-called supernode. Supernodes are directory nodes which are enlarged by a multiple of the block size. The X-tree outperforms the R*-tree by a factor of up to 400 for point queries.

All these approaches have in common that they must use the 50%-quantile when splitting a data page in order to fulfill storage utilization guarantees. As we will show in the next Section, this is the worst case in high-dimensional indexing, because the resulting pages have an access probability close to 100%.

To overcome this drawback, Berchtold, Böhm and Kriegel recently proposed another approach in [3] where they applied unbalanced partitioning of space. The proposed technique is an efficient bulk-loading operation of an X-tree. However, the approach is applicable only if all the data is known *a priori* which is not always the case. Additionally, due to restrictions of the X-tree directory, a peel-like partitioning cannot be achieved which is important for indexing high-dimensional data spaces, as we will see.

3 Analysis of Balanced Splits

It is well-known that for low-dimensional indexes it is beneficial to minimize the perimeter of the bounding boxes of the page regions so that all sides of the bounding box have approximately the same length [9]. Such space partitioning is usually achieved by recursively splitting the data space into equally filled regions i.e. at the 50%-quantile. Therefore, we call such a split strategy "*balanced split*". In the following cost model, we assume a database of N objects in a d-dimensional data space. The points are uniformly distributed in the unit hypercube $[0, 1]^d$. As we will show in the experimental part, our results are also valid for real data which are correlated and clustered. Further, we assume hypercubes with side-length q as queries, which are taken randomly from the data space.

In high-dimensional spaces, some unexpected effects lead to performance degeneration when applying a balanced split. For a more detailed description of these effects we refer the reader to [5]. The first observation is that, at least when applying balanced partitioning to a uniformly distributed data set, the data space cannot be split in each dimension. For example, assuming a 20-dimensional data space which has been split exactly once in each dimension, would require $2^{20} \approx 1,000,000$ data pages or 30,000,000 data objects if the average page occupancy is 30 objects. Following the notation used in the literature we will call the average page occupancy effective page capacity C_{eff}. The data space is usually split only once in a number d' of dimensions and is not split in the remaining $(d - d')$ dimensions. Thus, the bounding boxes of the page regions include almost the whole extension of the data space in these dimensions. If we assume the data space to be the d-dimensional unit hypercube $[0, 1]^d$, the bounding boxes have approximately side length 1/2 in d' dimensions and approximately side length 1 in $(d - d')$ dimensions. The number d' of dimensions, splitting the data space exactly once can be determined from the number N of objects stored in the database and the effective page capacity, as follows:

$$d' = \log_2\left(\frac{N}{C_{eff}}\right).$$

The second observation is that a similar property holds for typical range queries. If we assume that the range query is a hypercube and should have a selectivity s, then the side length q equals to the d-th root of s: $q = \sqrt[d]{s}$. For a 20-dimensional range query with selectivity 0.01% we obtain a side length $q = 0.63$ which is larger than half of the extension of the data space in this dimension.

It becomes intuitively clear that a query with side length larger than 0.5 must intersect with every bounding box having at least side length 0.5 in each dimension. However, we are able to model this effect more accurate: The performance of a multi-dimensional range query is usually modeled by means of the so-called Minkowski sum which transforms the range query into an equivalent point query by enlarging the bounding boxes of the pages accordingly [5]. In low-dimensional spaces, usually so-called boundary effects are neglected i.e., the data space is assumed to be infinite and everywhere filled with objects accord-

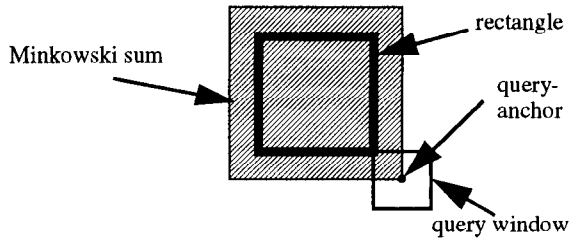

Figure 1: The Minkowski Sum

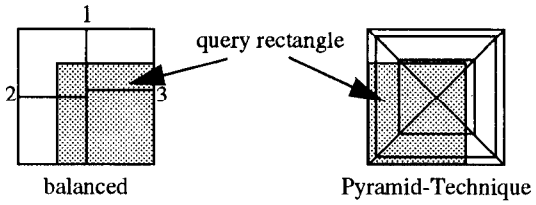

Figure 3: Partitioning Strategies

ing to the same density and therefore, no objects intersect the boundary of the data space.

To determine the probability that the bounding box of a page region intersects the query region, we consider the portion of the data space in which the center point of the query must be located such that query and bounding box intersect. Therefore, we move the center point of the query, the query anchor, to each point of the data space marking the positions where the query rectangle intersects the bounding box (c.f. Figure 1). The resulting set of marked positions is called the Minkowski sum which is the original bounding box having all sides enlarged by the query side length q. Taking into account that the volume of the data space is 1, the Minkowski sum directly corresponds to the intersection probability. In practice, often a corner of the query rather than the center is used as query anchor. Let $LLC_{i,j}$ and $URC_{i,j}$ denote the j-th coordinates of the "lower left" and "upper right" corner of bounding box i ($0 \le i < N$, $0 \le j < d$). The expected value $P_{\text{no\_bound\_eff}}(q)$ for page accesses when processing a range query with side length q then is

$$P_{\text{no\_bound\_eff}}(q) = \sum_{i} \prod_{j=0}^{d-1} (\text{URC}_{i,j} - \text{LLC}_{i,j} + q)$$

In order to adapt this formula to boundary effects, especially considering that the bounding boxes as well as the query hypercubes are always positioned completely in the data space, we obtain:

$$P_{\text{bound\_eff}}(q) = \sum_{i} \prod_{j=0}^{d-1} \frac{\min(\text{URC}_{i,j}, 1-q) - \max(\text{LLC}_{i,j} - q, 0)}{1-q}$$

The minimum and maximum are necessary to cut the parts of the Minkowski sum exceeding the data space, whereas the denominator $(1 - q)$ is due to fact that the stochastic "event space" of the query anchor is not [0, 1] but rather [0, 1-q]. The model for balanced splits can be simplified if the number of data pages is a power of two. Then, all pages have the extension 0.5 in d' dimensions, accommodated in the lower or the upper half of the data space, and full extension in the remaining dimensions. By C_{eff} we denote the effective (average) capacity of a data page. It is dependent on d. As in our special case, all pages have the same access probability and thus, the expected value of data page accesses is:

$$E_{\text{balanced}}(d, q, N) = \frac{N}{C_{eff}(d)} \cdot \min\left(1, \left(\frac{0.5}{1-q}\right)^{\log_2\left(\frac{N}{C_{eff}(d)}\right)}\right)$$

Note that we require the minimum to assure that the expected value doesn't exceed the total number of data pages and that we are able to ignore the remaining $(d - d')$ dimensions because the extension of the data pages in these dimensions is 1.

Figure 2 depicts the cost of range query processing using balanced splits, as estimated by our model. In this figure, the dimension is varied, whereas the database size and the selectivity of the query is constant. The percentage of accessed pages quickly approaches the 100%-mark which is actually met at dimension 10. Efficient query processing is only possible in dimensions less than 8.

This performance degeneration is a problem of all index structures which strive for a split at or close to the 50%-quantile of a data set. The only way around this dilemma is to split in an unbalanced way. Figure 3 depicts the partitions resulting from a balanced and a peel-like split of the Pyramid-Technique in a 2-dimensional example. As depicted, a large range query will intersect all of the partitions when splitting in a balanced way, but only a few pages, when splitting in peels. Besides, in the 2-dimensional example the effect, that most pages are intersected by the query can only be seen for a maximum of four pages. When going to higher dimensions, e.g. to a 5-dimensional space, then $2^5 = 32$ pages can be created by splitting in each dimension exactly once. In this case, all 32 pages are accessed. In contrast, the pyramid technique yields 10 pyramids in the 5-dimensional data space. Each pyramid is partitioned into three or 4 pieces. Like in the 2-dimensional example, some of the partitions are very likely not to be intersected by the query (In our figure, half of the pyramids are scanned completely. In the other half, only one out of three partitions are read. Together, 10 page accesses are saved. This effect becomes stronger with increasing dimension).

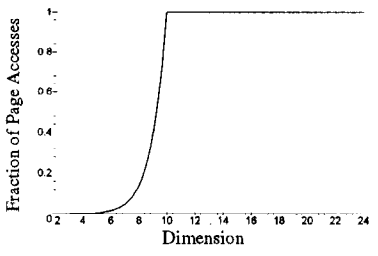

Figure 2: Estimated Cost of Query Proc. Using the X-Tree

4 The Pyramid-Technique

The basic idea of the Pyramid-Technique is to transform the d-dimensional data points into 1-dimensional values and then store and access the values using an efficient index structure such as the B^+-tree [1, 10]. Potentially, any order-preserving one-dimensional access method can be used. Operations such as insert, update, delete or search operations are performed using the B^+-tree. Figure 4 depicts the general procedure of an insert operation and the processing of a range query. In both cases, the d-dimensional input is transformed into some 1-dimensional information which can be processed by the B^+-tree. Note that, although we index our data using a 1-dimensional key, we store d-dimensional points *plus* the corresponding 1-dimensional key in the leaf nodes of the B^+-tree.

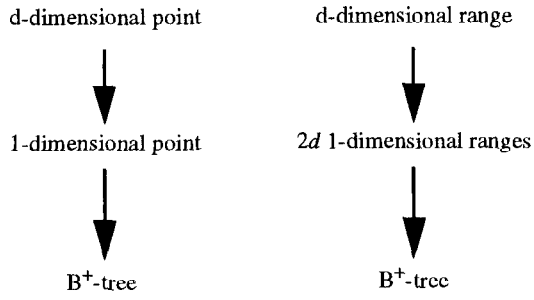

Figure 4: Operations on Indexes

Therefore, we do not have to provide an inverse transformation. The transformation itself is based on a specific partitioning of the data space into a set of d-dimensional pyramids. Thus, in order to define the transformation, we first explain the data space partitioning of the Pyramid-Technique.

4.1 Data Space Partitioning

The Pyramid-Technique partitions the data space in two steps: in the first step, we split the data space into $2d$ pyramids having the center point of the data space (0.5, 0.5, ..., 0.5) as their top and a $(d\text{-}1)$-dimensional surface of the data space as their base. In a second step, each of the $2d$ pyramids is divided into several partitions each corresponding to one data page of the B$^+$-tree. In the 2-dimensional example depicted in Figure 5, the space has been divided into 4 triangles (the 2-dimensional analogue of the d-dimensional pyramids) which all have the center point of the data space as top and one edge of the data space as base (Figure 5 left). In a second step, these 4 partitions are split again into several data pages parallel to the base line (Figure 5 right). Given a d-dimensional space instead of the 2-dimensional space, the base of the pyramid is not a 1-dimensional line such as in the example, but a $(d\text{-}1)$-dimensional hyperplane. As a cube of dimension d has $2d$ $(d\text{-}1)$-dimensional hyperplanes as a surface, we obviously obtain $2d$ pyramids.

Numbering the pyramids as in the 2-dimensional example in Figure 6a, we can make the following observations which are the basis of the partitioning strategy of the Pyramid-Technique: All points located on the i-th $(d\text{-}1)$-dimensional surface of the cube (the base of the pyramid) have the common property that either their i-th coordinate is 0 or their $(i-d)$-th coordinate is 1. We observe that the base of the pyramid is a $(d - 1)$-dimensional hyperplane, because one coordinate is fixed and $(d - 1)$ coordinates are variable. On the other hand, all points v located in the i-th pyramid p_i have the common property that the distance in the i-th coordinate from the center point is either smaller than the distance of all other coordinates if $i < d$, or larger if $i \geq d$. More formally:

$$\forall j,\ 0 \leq j < d, j \neq i:\ (|0.5 - v_i| \leq |0.5 - v_j|) \qquad if(i < d)$$
$$\forall j,\ 0 \leq j < d, j \neq (i-d):\ (|0.5 - v_{(i-d)}| \geq |0.5 - v_j|)\ \ if(i \geq d)$$

Figure 6b depicts this property in two dimensions: all points located in the lower pyramid are obviously closer to the center point in their d_0-direction than in their d_1-direction. This common property provides a very simple way to determine the pyramid p_i which includes a given point v: we only have to determine the dimension i having the maximum deviation $|0.5 - v_i|$ from the center. More formally:

Definition 1: ***(Pyramid of a point v)***
A d-dimensional point v is defined to be located in pyramid p_i,

$$i = \begin{cases} j_{max} & if(v_{j_{max}} < 0.5) \\ (j_{max} + d) & if(v_{j_{max}} \geq 0.5) \end{cases}$$

$$j_{max} = (j | (\forall k,\ 0 \leq (j,k) < d, j \neq k:\ |0.5 - v_j| \geq |0.5 - v_k|))$$

Note that all further considerations are based on this definition which therefore is crucial for our technique.

Another important property is the location of a point v within its pyramid. This location can be determined by a single value which is the distance from the point to the center point according to dimension j_{max}. As this geometrically corresponds to the height of the point within the pyramid, we call this location height of v (c.f. Figure 7)

Definition 2: ***(Height of a point v)***
Given a d-dimensional point v. Let p_i be the pyramid in which v is located according to Definition 1. Then, the height h_v of the point v is defined as

$$h_v = |0.5 - v_{i \text{ MOD } d}|$$

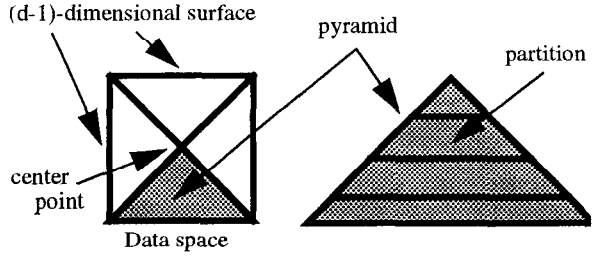

Figure 5: Partitioning the Data Space into Pyramids

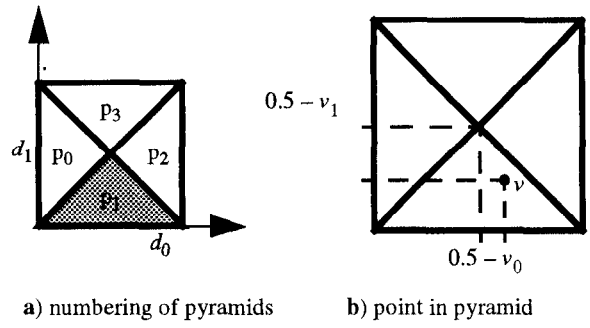

Figure 6: Properties of Pyramids

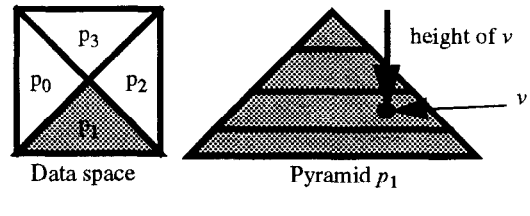

Figure 7: Height of a Point within it's Pyramid

Using Definition 1 and Definition 2, we are able to transform a d-dimensional point v into a value $(i+h_v)$ where i is the index of the according pyramid p_i and h_v is the height of v within p_i. More formally:

Definition 3: ***(Pyramid value of a point v)***
Given a d-dimensional point v. Let p_i be the pyramid in which v is located according to Definition 1 and h_v be the height of v according to Definition 2. Then, the pyramid value pv_v of v is defined as

$$pv_v = (i + h_v)$$

Note that i is an integer and h_v is a real number in the range [0, 0.5]. Therefore, each pyramid p_i covers an interval of $[i, (i+0.5)]$ pyramid values and the sets of pyramid values covered by any two pyramids p_i and p_j are disjunct. Note further that this transformation is not injective i.e., two points v and v' may have the same pyramid value. But, as mentioned above, we do not require an inverse transformation and therefore we do not require a bijective transformation.

4.2 Index Creation

Given the transformation determining the pyramid value of a point q, it is a simple task to build an Index based on the Pyramid-Technique. In order to dynamically insert a point v, we first determine the pyramid value pv_v of the point and insert the point into a B$^+$-tree using pv_v as a key. Finally, we store the d-dimensional point v and pv_v in the according data page of the B$^+$-tree. Update and delete operations can be done analogously. Note that B$^+$-trees can be bulk-loaded very efficiently e.g, when building a B$^+$-tree from a large set of data items. The bulk-loading techniques for B$^+$-trees can be applied to the Pyramid-Technique, as well.

In general, the resulting data pages of the B$^+$-tree contain a set of points which all belong to the same pyramid and have the common property that their pyramid value lies in an interval given by the minimal and maximal key value of the data pages. Thus, the geometric correspondence of a single B$^+$-tree data page is a partition of a pyramid as shown in Figure 7 (right).

5 Query Processing

In contrast to the insert, delete and update operation, query processing using the Pyramid-Technique is a complex operation. Let us focus on point queries first which are defined as *"Given a query point q, decide whether q is in the database"*. Using the Pyramid-Technique, we can solve the problem by first computing the pyramid value pv_q of q and querying the B$^+$-tree using pv_q. As a result, we obtain a set of d-dimensional points sharing pv_q as a pyramid value. Thus, we scan the set and determine whether the set contains q and output the result.

In case of range queries, the problem is defined as follows: *"Given a d-dimensional interval*

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

determine the points in the database which are inside the range." Note that the geometric correspondence of a multidimensional interval is a hyper-rectangle. Analogously to point queries, we face the problem to transform the d-dimensional query into a 1-dimensional query on the B$^+$-tree. However, as the simple 2-dimensional example depicted in Figure 8 (left) demonstrates, a query rectangle may intersect several pyramids and the computation of the area of intersection is not trivial. As we also take from the example, we first have to examine which pyramids are

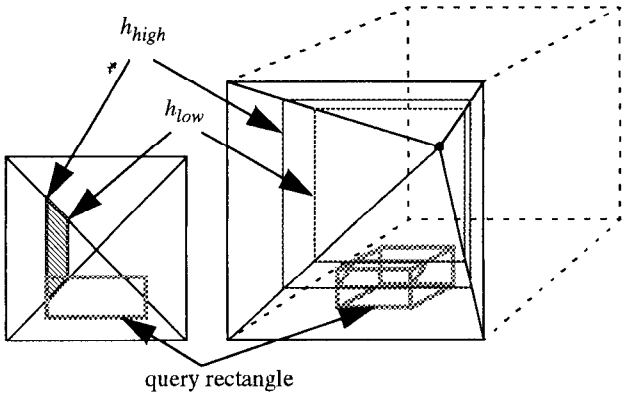

Figure 8: Transformation of Range Queries

affected by the query, and second, we have to determine the ranges inside the pyramids. The test whether a point is inside the ranges is based on a single attribute criterion (h_v between two values). Therefore, determining all such objects is a one-dimensional indexing problem. Objects outside the ranges are guaranteed not to be contained in the query rectangle. Points lying inside the ranges, are candidates for a further investigation. It can be seen in Figure 8 that some of the candidates are hits, others are false hits. Then, a simple point-in-rectangle-test is performed in the refinement step.

For simplification, we focus the description of the algorithm only on pyramids p_i where $i < d$, however, our algorithm can be extended to all pyramids in a straight-forward way. As a first step of our algorithm, we transform the query rectangle q into an equivalent rectangle $\hat{q}$ such that the interval is defined relative to the center point.

$$\hat{q}_{j_{min}} = q_{j_{min}} - 0.5 \text{ and } \hat{q}_{j_{max}} = q_{j_{max}} - 0.5 \ , \ \forall j, 0 \le j < d$$

Additionally, we interpret any point v mentioned in this section to be defined relative to the center point of the data space. Based on Definition 1, we are able to determine if a pyramid p_i is affected by a given query $\hat{q}$. As we will see, we have to determine the absolute minimum and maximum of an interval which is defined as follows: Let $MIN(r)$ be defined as the minimum of the absolute values of an interval r:

$$MIN(r) = \begin{cases} 0 & \text{if } r_{min} \le 0 \le r_{max} \\ min(|r_{min}|, |r_{max}|) & \text{otherwise} \end{cases}$$

Note that $|r_{min}|$ may be larger than $|r_{max}|$. Analogously, we define

$$MAX(r) = max(|r_{min}|, |r_{max}|)$$

Lemma 1: ***(Intersection of a Pyramid and a Rectangle)***
A pyramid p_i is intersected by a hyperrectangle $[\hat{q}_{0_{min}}, \hat{q}_{0_{max}}], \ldots, [\hat{q}_{d-1_{min}}, \hat{q}_{d-1_{max}}]$ if and only if

$$\forall j, 0 \le j < d, j \ne i: \hat{q}_{i_{min}} \le -MIN(\hat{q}_j)$$

Proof:

The query rectangle intersects pyramid p_i, iff there exists a point v inside the rectangle which falls into pyramid p_i. Thus, the coordinates $|v_j|$ of v must all be smaller than $|v_i|$. This, however, is only possible if the minimum absolute value in the query rectangle in dimension j is closer to the center point than $\hat{q}_{i_{min}}$ is to

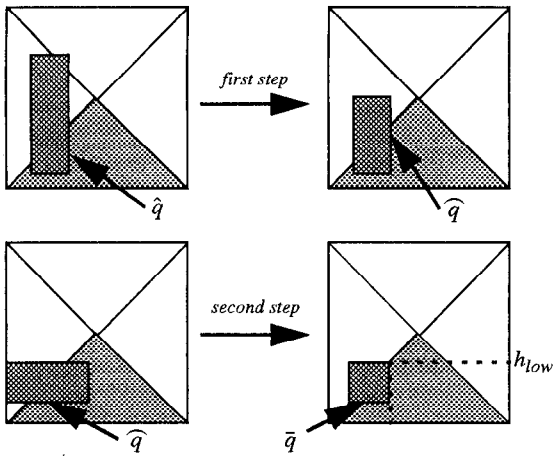

Figure 9: Restriction of Query Rectangle

```
Point_Set PyrTree::range_query(range q)
{
  Point_Set res;
  for (i = 0; i < 2d; i++) {
    if (intersect(p[i], q) {
    // using Lemma 1
      determine_range(p[i], q, h_low, h_high);
      // using Lemma 2

      cs = btree_query(i+h_low, i+h_high);
      for (c = cs.first; cs.end; cs.next){
        if (inside(q, c))
          res.add(c);
      }
    }
  }
  return res;
}
```

Figure 10: Processing Range Queries (Algorithm)

the center point. Lemma 1 follows from the fact that this must hold for all dimensions j. **q.e.d.**

In the second step, we have to determine which pyramid values inside an affected pyramid p_i are affected by the query. Thus, we are looking for an interval $[h_{low}, h_{high}]$ in the range of [0, 0.5] such that the pyramid values of all points inside the intersection of the query rectangle and pyramid p_i are in the interval $[i+h_{low}, i+h_{high}]$. Figure 8 depicts this interval for two and three dimensions.

In order to determine h_{low} and h_{high}, we first restrict our query rectangle to pyramid p_i i.e., we remove all points above the center point:

$$\widehat{q}_{i_{min}} = \hat{q}_{i_{min}},\ \widehat{q}_{i_{max}} = min(\hat{q}_{i_{max}}, 0),$$

$$\widehat{q}_{j_{min}} = \hat{q}_{j_{min}}, \text{ and } \widehat{q}_{j_{max}} = \hat{q}_{j_{max}}, \text{ where } (0 \le j < d), j \ne i .$$

Note that we restricted our considerations to the pyramids $p_0 \ldots p_{d-1}$. Therefore, the relevant values of $\hat{q}_{i_{min}}$ and $\hat{q}_{i_{max}}$ are negative. The effect of this restriction is depicted in a two-dimensional example in Figure 9 (upper).

The determination of the interval $[h_{low}, h_{high}]$ is very simple if the center point of the data space is included in the query rectangle i.e., $\forall j, (0 \le j < d)$: $(\hat{q}_{j_{min}} \le 0 \le \hat{q}_{j_{max}})$. In this case, we simply use the extension of the query rectangle as a result, thus:

$$h_{low} = 0 \text{ and } h_{high} = MAX(\widehat{q}_i).$$

If the center point is not included in the query rectangle, we first make the observation that $h_{high} = MAX(\widehat{q}_i)$, too. This is due to the fact that the query rectangle must contain at least one point v such that $v_i = MAX(\widehat{q}_i)$ because otherwise there would be no intersection between the query rectangle and pyramid p_i.

In order to find the value h_{low}, we have to determine the minimum height of points inside the query rectangle and the pyramid p_i. As we consider points which are inside $\widehat{q}$ and inside p_i, we can intersect all intervals $[\widehat{q}_{j_{min}}, \widehat{q}_{j_{max}}]$ $(0 \le j < d), j \ne i$ with $[\widehat{q}_{i_{min}}, \widehat{q}_{i_{max}}]$ without affecting the value h_{low}. Then, the minimum of the *min*-values of all dimensions of the new rectangle $\bar{q}$ equals to h_{low}. Figure 9 (lower) shows an example of this operations. Obviously, the checkered rectangles on the left and the right side of each example are causing the same value h_{low}.

Lemma 2: ***(Interval of Intersection of Query and Pyramid)***
Given a query interval $\hat{q}$ and an affected pyramid p_i, the intersection interval $[h_{low}, h_{high}]$ is defined as follows:

Case 1: ($\forall j, 0 \le j < d$: $(\hat{q}_{j_{min}} \le 0 \le \hat{q}_{j_{max}})$)

$$h_{low} = 0$$

$$h_{high} = MAX(\widehat{q}_i)$$

Case 2: (otherwise)

$$h_{low} = min_{(0 \le j < d, j \ne i)}(\bar{q}_{j_{min}}) \quad (*)$$

$$h_{high} = MAX(\widehat{q}_i)$$

$$\bar{q}_{j_{min}} = \begin{cases} max(MAX(\widehat{q}_i), MIN(\widehat{q}_j)) & \text{if } MAX(\widehat{q}_j) \ge MIN(\widehat{q}_i) \\ MIN(\widehat{q}_i) & \text{otherwise} \end{cases}$$

Proof:

We will show for any point v which is located inside the query rectangle $\hat{q}$ and an affected pyramid p_i that the resulting query interval $[h_{high}, h_{low}]$ contains $|v_i|$. Note that we assumed i to be smaller than d and thus $v_i < 0$. Therefore, we have to show that

$$h_{low} \le |v_i| \le h_{high}.$$

1. $|v_i| \le h_{high}$:

This holds because we chose h_{high} such that $|v_i| \le MAX(\widehat{q}_i) = h_{high}$.

2. $h_{low} \le v_i$:

If $\hat{q}$ contains the center point, we have $h_{low} = 0 \le |v_i|$.

Otherwise, $|v_i| \ge |v_j|, (\forall j, (0 \le j < d))$ because v is inside the pyramid i. On the other hand, $v_j \ge \hat{q}_{j_{min}}, \forall j, (0 \le j < d)$ because v is inside the query rectangle and $v_j \ge \widehat{q}_{j_{min}}$ because all coordi-

nates of v are negative for $0 \le i < d$. Thus, $|v_j| \ge MIN(\widehat{q}_j), (\forall j, 0 \le j < d)$.

Additionally, $|v_i| \ge MIN(\widehat{q}_i)$ because of the same reasons. Assembling the two results, we derive: $|v_i| \ge max(MIN(\widehat{q}_i), MIN(\widehat{q}_j)), \forall j, 0 \le j < d$. From equation (*), however, follows that $\bar{q}_{j_{min}} \ge h_{low}$. So we finally obtain that $|v_i| \ge max(MIN(\widehat{q}_i), MIN(\widehat{q}_j)) \ge h_{low}$ **q.e.d.**

Lemma 1 and Lemma 2 imply the simple query processing algorithm depicted in Figure 10.

6 Analysis of the Pyramid-Technique

For this analysis, we assume a uniform distribution of the data space and of the query hypercubes. We propose a cost model for the Pyramid-Technique, comparable to the model in section 3, to analytically show the superiority of the Pyramid-Technique. Thus, we model the cost for processing hypercube shaped range queries having a side length larger than 0.5 to achieve a reasonable selectivity for high-dimensional queries. In this case, the center of the data space is always contained in the query and therefore, our window query is transformed into a set of exactly $2d$ one-dimensional range queries with,

$$h_{low} = 0 \text{ and } h_{high} = MAX(\widehat{q}_i).$$

We do not need the concept of the Minkowski sum here because we analyze the performance of one-dimensional interval queries. However, we have to take into account that, in contrast to the points of the database, the pyramid values are not uniformly distributed.

In the first step of our model, we determine an expected value for the amount of data in each pyramid, which has to be accessed during query processing (the size of the candidate set). We consider the lower left corner of the query $QA = (q_{0_{min}}, ..., q_{d-1_{min}})$ as the anchor point of the query. QA is obviously taken from the multidimensional interval $QAI = [0, 1-q]^d$ to guarantee that the whole query is located inside the data space. Therefore, the height h_{high} in pyramid p_i is uniformly distributed in the interval $H_i = [q - 0.5, 0.5]$ (c.f. Figure 11). We call the part of the hyper-pyramid, starting with $h_{low} = 0$ and ending with h_{high} (underlaid in grey in Figure 11) the affected part of the pyramid. The volume of affected part can be determined using the fact that it is the $2d$-th part of a hypercube with side length $2 \cdot h_{high}$:

$$V(h_{high}) = \frac{(2 \cdot h_{high})^d}{2 \cdot d}.$$

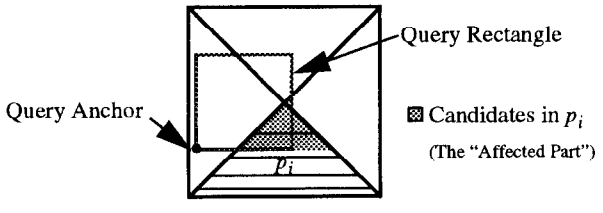

Figure 11: Modeling the Pyramid-Technique

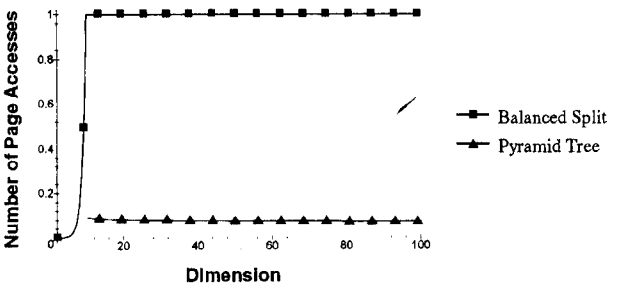

Figure 12: Number of page accesses when processing range queries for the Pyramid Tree and Balanced Splitting

From this volume of the affected part for a given h_{high}, we can also determine the expected value by forming an average over all possible positions of h_{high} in the interval H_i. Thus, we have to integrate over h_{high} and then divide the result by the size of the interval H_i, which yields the following integral formula:

$$E_V(d, q) = \frac{\int_{q-0.5}^{0.5} V(h_{high}) \; d\, h_{high}}{0.5 - (q - 0.5)}$$

The integral can be evaluated and simplified to:

$$E_V(d, q) = \frac{1 - (2q-1)^{d+1}}{4d \cdot (1-q) \cdot (d+1)}$$

As $E_V(d, q)$ is the expected volume of the affected part for a query of size q in a single pyramid, under the uniformity assumption, $2d \cdot E_V(d, q) \cdot (N/2d) = E_V(d, q) \cdot N$ is the expected total number of objects in the affected parts of all pyramids.

These objects are the candidates for an exact-geometry test of d-dimensional range containment (c.f. Figure 11). Since it is unlikely that the affected part is perfectly aligned with a break between two subsequent pages, the question is, how many data pages are occupied by the candidates. Note that all candidates belong to a single interval of pyramid values and therefore, the candidates are stored contiguously on the data pages. Thus, assuming a pagination with the effective page capacity C_{eff}, we have to descend the directory of the B$^+$-tree for each pyramid to find the object with the lowest pyramid value in each pyramid. This object may be located anywhere inside a data page. Then, we have to read a run with the length of $E_V(d, q) \cdot N$ objects, which occupies $E_V(d, q) \cdot N / C_{eff}$ data pages. The last object is, again located somewhere on a data page with an equal probability of every position on the page. On average, we have to read half a page before and after the run, respectively. Therefore, the required number of accesses to data pages for all $2d$ pyramids is:

$$E_{\text{pyramidtree}}(d, q, N) = \frac{2d + N \cdot (1 - (2q-1)^{d+1})}{2C_{eff}(d) \cdot (d+1) \cdot (1-q)}$$

The number of accesses to directory pages is $2d$ times the height of the B$^+$-tree $\log_{C_{eff,dirpg}}(N/C_{eff})$ and can be neglected because the directory fits into the cache. We made the same assumption in the model for balanced splitting. Figure 12 depicts the performance of the Pyramid-Technique as predicted by our model and, in comparison, the estimated cost when using balanced splitting. The Pyramid-Technique does not reveal any performance degeneration in high dimensions.

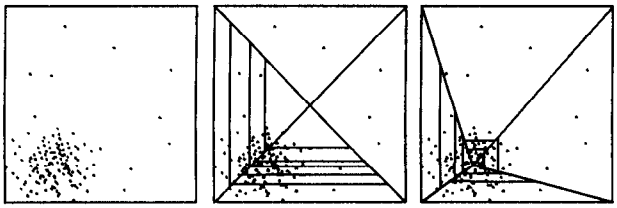

Figure 13: Effect of Clustered Data

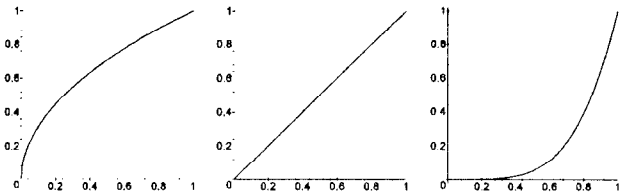

Figure 14: Transformation Functions t_i

Note that we achieved this result by assuming hypercube shaped queries, which are uniformly distributed over the data space and, therefore, the result only holds for this query type.

7 The Extended Pyramid-Technique

All our considerations presented so far were based on the assumption that the data is uniformly distributed. However, data produced by real-life applications does not behave this way. Therefore, the question arises, how to adapt the Pyramid-Technique to real data. Let us consider the following scenario: What happens to the Pyramid-Technique if most of the data is located in one corner of the data space (Figure 13 left). Obviously, only a few pyramids (in the extreme case only one) will contain most of the data while the other pyramids are nearly empty. This, however, will result in the suboptimal space partitioning depicted in the example in Figure 13 (middle). Obviously, partitioning is suboptimal because we can assume real-life queries to be similarly distributed as the data itself. Under this realistic assumption, a much better partitioning for the same data set is shown in Figure 13 (right).

The basic idea of the extended Pyramid-Technique is to achieve such a partitioning by transforming the data space such that the data cluster is located in the center point (0.5, ..., 0.5) of space. Thus, we have to map the given data space to the canonical data space $[0, 1]^d$ such that the d-dimensional median of the data is mapped to the center point. Note that we only have to assure that the median of the data roughly coincides with the center point of the data space. The presence of clusters distributed over the space does not cause a problem for our technique. However, we only apply the transformation to determine the pyramid values of points and query rectangles, but not to the points itself. Therefore, we do not have to apply the inverse transformation to our answer set.

As the computation of the d-dimensional median is a hard problem, we use the following heuristic to determine an approximation of the d-dimensional median: We maintain a histogram for each dimension to keep track of the median in this dimension. The d-dimensional median is then approximated by the combination of the d one-dimensional medians. Obviously, the approximated d-dimensional median may be located outside the convex hull of the data cluster. As our experiments showed, this effect occurs very rarely and therefore the performance of our algorithms is not affected. The computation of the median can either be done dynamically in case of dynamic insertions, or once in case of a bulk-load of the index.

Given the d-dimensional median mp of the data set, we define a set of d functions t_i, $0 \le i < (d-1)$ transforming the given data space in dimension i such that the following conditions hold:

1. $t_i(0) = 0$

2. $t_i(1) = 1$

3. $t_i(mp_i) = 0.5$

4. t_i: $[0, 1] \rightarrow [0, 1]$

The three conditions are necessary to assure that the transformed data space still has an extension of $[0..1]^d$ (1. and 2.), and that the median of the data becomes the center point of the data space (3.). Condition 4. assures that each point in the original data space is mapped to a point inside the canonical data space. The resulting functions t_i can be chosen as an exponential function such that:

$$t_i(x) = x^r$$

Obviously, conditions 1., 2., and 4. are satisfied by x^r, $r \ge 0$, $0 \le x \le 1$. In order to determine the parameter r, we have to satisfy condition 3: $t_i(mp_i) = 0.5 = mp_i^r$. Thus, $r = \frac{1}{-\log_2(mp_i)}$ and

$$t_i(x) = x^{\frac{1}{-\log_2(mp_i)}}$$

Now, in order to insert a point v into an index using the extended Pyramid-Technique, we simply transform v into a point $v'_i = t_i(v_i)$ and determine the pyramid value $pv_{v'}$. Then, we insert v using $pv_{v'}$ as a key value as described in section 4.2. In order to process a query, we first transform the query rectangle q (or query point) into a query rectangle q' such that $q'_{i_{min}} = t_i(q_{i_{min}})$ and $q'_{i_{max}} = t_i(q_{i_{max}})$. Note that q' is a rectangle because we applied independent transformations to each dimension. Next, we use the algorithm presented in section 5, to determine the intervals of affected pyramid values and query the B^+-tree. As a result, we obtain a set of non-transformed d-dimensional points v which we test against the original query rectangle q. Note that we used the transformations t_i only to determine the pyramid value but we have not transformed the points itself.

If we dynamically build an index, the situation may occur that the first 10% of inserted points have a median different from that of the other 90% of the data. More general, we have to handle the situation that the median changes during the insertion process. To handle this case, we maintain the current median by maintaining a histogram for each dimension and re-build the index, if the distance of the current median to the center point exceeds a certain threshold. Note that re-building the index is not too expensive because we make use of a bulk-load technique for B^+-trees. In order to determine a good threshold, we use the value $th = (\sqrt{d})/4$ because the maximum distance from any point to the center point is $(\sqrt{d})/2$ and therefore, the adapting process is guaranteed to terminate after a logarithmic number of steps. Note

further that the probability that the median shifts and therefore the index has to be reorganized decreases with an increasing percentage of inserted data items. Therefore, a reorganization occurs very rarely in practice. Furthermore, our experiments showed that a slightly shifted median has a negligible influence on the performance of the Pyramid-Technique.

8 Experimental Evaluation

To demonstrate the practical impact of the Pyramid-Technique and to verify our theoretical results, we performed an extensive experimental evaluation of the Pyramid-Technique and compared it to the following competitive techniques:

- X-tree [6]
- Hilbert-R-tree [13]
- Sequential Scan.

The Hilbert-R-tree has been chosen for comparison, because the Hilbert-curve and other space filling curves can be used in conjunction with a B-tree in a so-called one-dimensional embedding. Since the Pyramid-Technique also incorporates a very sophisticated one-dimensional embedding, the Hilbert R-tree appeared to us as a natural competitive method.

Recently, the criticism arose that index-based query processing is generally inefficient in high-dimensional data spaces [8], and that sequential scan processing yields better performance in this case. Therefore, we included the sequential scan in our experiments. We will confirm the observation that the sequential scan outperforms the X-Tree and the Hilbert R-Tree for high dimensionalities, but we will also see that our new technique outperforms the sequential scan over in all experiments performed.

For clarity, we state our assumption that all relevant information is stored in the various indexes, as well as in the file used for the sequential scan. Therefore, no additional accesses to fetch objects for presentation or further processing are needed in any of the techniques applied in our experiments.

Our experiments have been computed on HP-9000/780 workstations with several GigaBytes of secondary storage.

Our evaluation comprises both, real and synthetic data sets. In all experiments, we performed range queries with a defined selectivity because range queries serve as a basic operation for other queries such as nearest neighbor queries or partial range queries. The query rectangles are selected randomly from the data space such that the distribution of the queries equals the distribution of the data set itself and the query rectangles are fully included in the data space. Thus, in case of uniform data we used uniformly distributed hypercube shaped query rectangles.

8.1 Evaluation Using Synthetic Data

Our synthetic data set contains 2,000,000 uniformly distributed points in a 100-dimensional data space. The raw data file occupies 800 MBytes of disk space. The main advantage of uniformly distributed point sets is, that it is possible to scale down the dimensionality of the point set by projecting out some of the dimensions without affecting the semantics of the query. We created files with varying dimension and varying number of objects by projection and selection and constructed various indexes using these raw data files.

In our first experiment (c.f. Figure 15) we measured the performance behavior with varying number of objects. We performed range queries with 0.1% selectivity in a 16-dimensional data space and varied the database size from 500,000 to 2,000,000 objects. Unfortunately, using our implementation the Hilbert-R-tree could only be constructed for a maximum of 1,000,000 objects due to limited main memory. The file sizes of all indexes in this experiment sum up to 1.1 GigaBytes. The page size in this experiment was 4096 Bytes, leading to an effective page capacity of 41.4 objects per page in all index structures. Figure 15 shows the performance of query processing in terms of number of page accesses, absorbed CPU-time and finally the total elapsed time, comprising CPU time and time spent in disk i/o. The speed-up with respect to the number of page accesses seems to be almost constant and ranges between 9.78 and 10.91. The speed-up in CPU time is higher than the speed-up in page accesses, but is only slightly increasing with growing database sizes. The reason is that B^+-trees facilitate an efficient in-page search for matching objects by applying bisection or interval search algorithms. However, most important is the speed-up in total elapsed time. It starts with factor 53, increases quickly and reaches its highest value with the largest database: The Pyramid-Technique with 2 million objects performs range queries 879 times faster than the corresponding X-tree! Range query processing on B^+-trees can be performed much more efficient than on X-trees because large parts of the tree can be traversed efficiently by following the side links in the data pages. Moreover, long-distance seek operations inducing expensive disk head movements have a lower probability due to better disk clustering possibilities in B^+-trees. The bar diagram on the right side of Figure 15 summarizes the highest speed-up factors in this experiment.

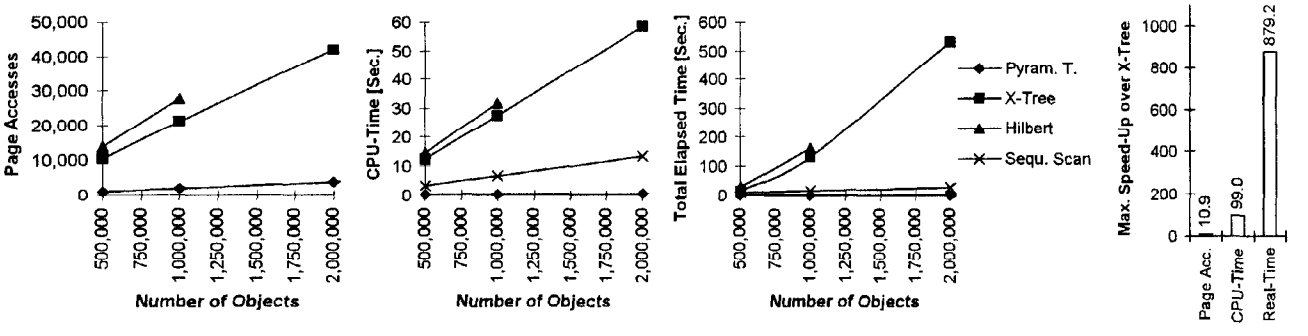

Figure 15: Performance Behavior over Database Size

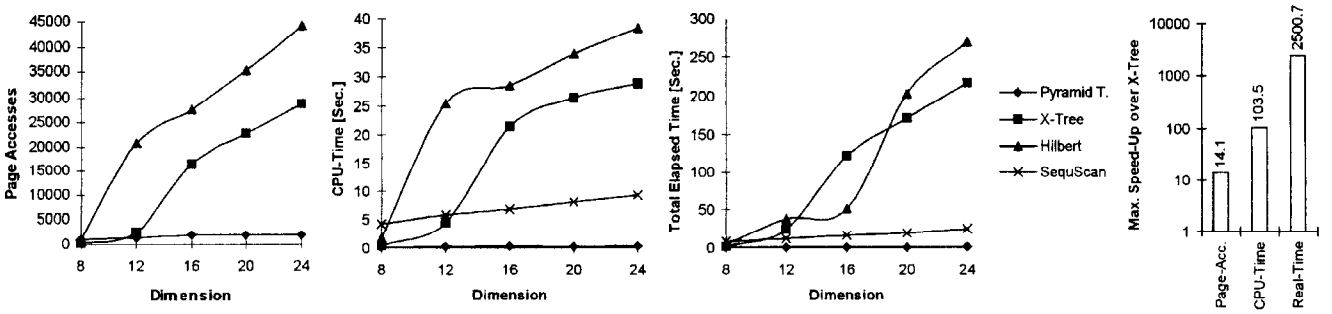

Figure 16: Performance Behavior over Data Space Dimension

In a second experiment, visualized in Figure 16, we determined the influence of the data space dimension on the performance of query processing. For this purpose we created 5 data files as projections of the original data files with the dimensionalities 8, 12, 16, 20, and 24 (the database size in this experiment is 1,000,000 objects) and created the corresponding indexes. The total amount of disk space occupied by the index structures used in this experiment sums up to 1.6 GigaBytes. The page size in this experiment was again 4096 Bytes. The effective data page capacity depends on the dimension and ranged from 28 to 83 objects per page. We investigated range queries with a constant selectivity of 0.01%. For a constant selectivity, the query range varies according to the data space dimension.

We observed that the efficiency of query processing using the X-tree rapidly decreases with increasing dimension up to the point where large portions of the index are completely scanned (16-dimensional data space). From this point on, the page accesses are growing linearly with the index size. Even worse is the performance of the Hilbert R-tree. A comparable deterioration of the performance with increasing dimension is not observable when using the Pyramid-Technique. Here, the number of page accesses, the CPU and total elapsed time grow slower than the size of the data set. The percentage of accessed pages with respect to all data pages is even reduced with growing dimensions (decreasing from 7.7% in the 8-dimensional experiment to 5.1% in the 24-dimensional experiment). The experiment yields a speed-up factor over the X-tree of up to 14.1 for the number of page accesses, and 103.5 for the CPU time. Furthermore, the Pyramid-Technique is up to 2500.7 times faster in terms of total elapsed time than the X-tree.

To demonstrate this observation that the percentage of pages accessed by the Pyramid-Technique decreases when going to higher dimensions, we determined the percentage of data pages accessed during query processing when indexing very high dimensions. Figure 17 depicts the result of this experiment: The percentage drops from 8.8% in 20 dimensions to 8.0% in 100 dimensions.

8.2 Evaluation Using Real Data Sets

In this series of experiments, we used data sets from two different application domains, information retrieval and data warehousing to demonstrate the practical impact of our technique.

The first data set contains text descriptors, describing substrings from a large text database extracted from WWW-pages. These text descriptors have been converted into 300,000 points in a 16-dimensional data space and were normalized to the unit hypercube. We varied the selectivity of the range queries from 10^{-5} to 31% and measured the query execution time (total elapsed time). The result is presented in Figure 18 and confirms our earlier results on synthetic data that the Pyramid-Technique clearly outperforms the other index structures. The highest speed-up factor observed was 51. Additionally, the experiment shows that the Pyramid-Technique outperforms the competitive structures for any selectivity i.e., for very small queries as well as for very large queries.

In a last series of experiments, we analyzed the performance of the Pyramid-Technique on a data set taken from a real-life data warehouse. The relation we used has 13 attributes: 2 categorical, 5 integer, and 5 floating point attributes. There are some very strong correlations in some of the floating point attributes, some of the attributes follow a very skewed distribution, whereas

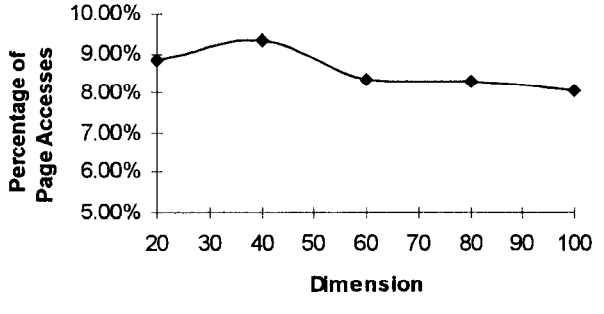

Figure 17: Percentage of Accessed Pages

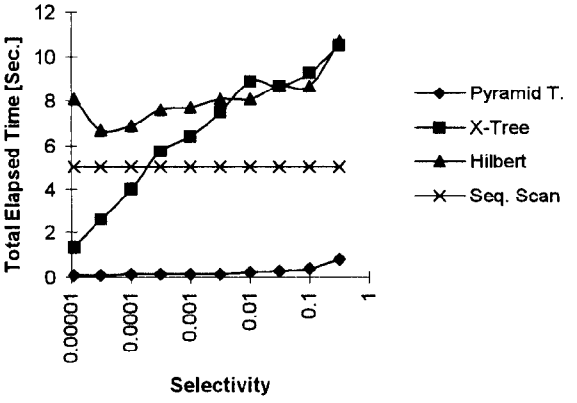

Figure 18: Query Processing on Text Data

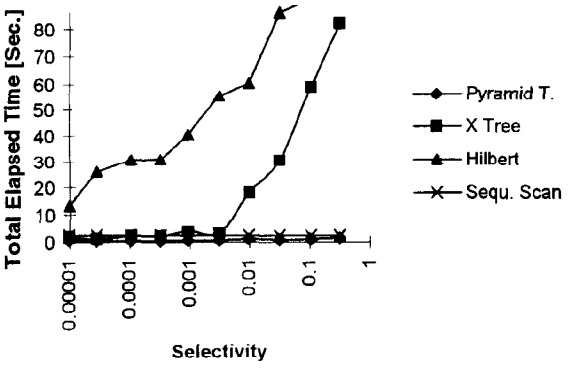

Figure 19: Query Processing on Warehousing Data

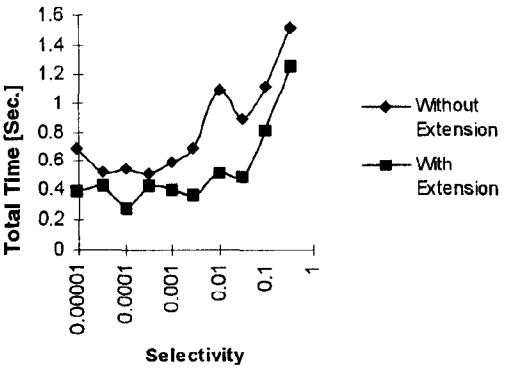

Figure 21: Performance of the Extended Pyramid-T.

some other attributes are rather uniformly distributed. The actual data set we used comprises a subset of 803,944 tuples containing data of a few months. In a first experiment, we measured the real time consumed during query processing. Again, the Pyramid-Technique outperformed the other index structures by orders of magnitude. As expected, the speed-up increases when going to higher dimensions because the effects described in section 3 apply more for larger query ranges. However, even for the smallest query range in the experiment, the speed-up factor over the X-tree was about 10.47, whereas the speed-up for the largest query range was about 505.18 in total query execution time.

In a second experiment, we measured the effect of the extension of the Pyramid-Technique proposed in section 7. We made the experiment on this data set because the data is very skew and the median is rather close to the origin of the data space in most of the dimensions. Figure 21 shows the effect of the extension. For all selectivities, there was a speed-up of about 10-40%. This shows first that for very skewed data, it is worth it reorganizing the index, and second that, if we refuse to do so, the loss of performance is not too high compared to the high speed-up factors over other index structures.

A major point of criticism is the argument that the Pyramid-Technique is designed for hypercube shaped range queries and might perform bad for other queries. Therefore, we ran an additional experiment investigating the behavior of the Pyramid-Technique for skewed queries. We generated partial range queries shrinking the data space in k dimensions and having the full

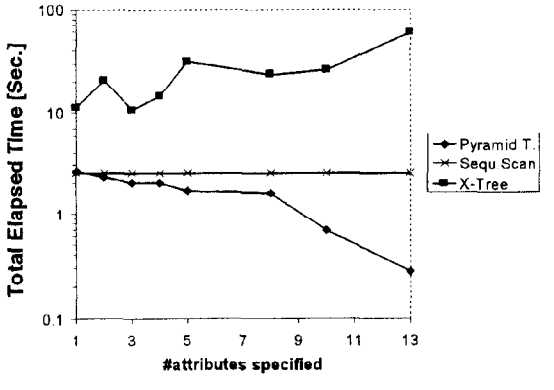

Figure 20: Varying the query mix (Warehouse Data)

extension of the data space in $(d\text{-}k)$ dimensions. These queries can be considered as $(d\text{-}k)$-dimensional hyper-slices in a d-dimensional space. As Figure 20 shows, the Pyramid-Technique outperforms the linear scan for all of these queries except the 1-dimensional queries. For 1-dimensional queries, the Pyramid-Technique required 2.6 sec. compared to 2.48 sec. for the linear scan. However, a large improvement was observed for 8-dimensional to 13-dimensional queries. The X-Tree couldn't compete with the Pyramid-Technique for any of these queries.

Summarizing the results of our experiments, we make the following observations:

1) For almost hypercube shaped queries, the Pyramid-Technique outperforms any competitive technique, including linear scan. This holds even for skewed, clustered and categorical data.

2) For queries having a bad selectivity, i.e. a high number of answers, or extremely skewed queries, especially queries specifying only a small number of attributes, the Pyramid-Technique still outperforms competitive index structures, however, a linear scan of the database is faster.

9 Conclusions

In this paper, we proposed a new indexing method, the Pyramid-Technique. It is based on a special partitioning strategy which has been optimized for high-dimensional range queries. The data space partitioning transforms d-dimensional points into 1-dimensional values which can be efficiently managed by a B^+-tree. We showed both, theoretically (assuming uniform distribution) as well as experimentally (for synthetic and real data) that the Pyramid-Technique outperforms other index structures such as the X-tree by orders of magnitude.

The concepts of the Pyramid-Technique come best into effect for hypercube shaped range queries. For very skewed queries or queries specifying only one attribute, the Pyramid-Technique performs worse than the linear scan. However, as our experiments show, none of the index structures proposed so far can handle very skewed queries efficiently. We plan to address the problem of handling strong skew in our future work.

References

[1] Bayer R., McCreight E.M.: *'Organization and Maintenance of Large Ordered Indices'*, Acta Informatica 1(3), 1977, pp. 173-189.

[2] Berchtold S., Böhm C., Braunmüller B., Keim D. A., Kriegel H.-P.: *'Fast Parallel Similarity Search in Multimedia Databases'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, Tucson, Arizona.

[3] Berchtold S., Böhm C., Kriegel H.-P.: *'Improving the Query Performance of High-Dimensional Index Structures Using Bulk-Load Operations'*, 6th. Int. Conf. on Extending Database Technology, Valencia, Spain, 1998.

[4] Berchtold S., Böhm C., Keim D., Kriegel H.-P., Xu X.:*'Optimal Multidimensional Query Processing Using Tree Striping'*, submitted.

[5] Berchtold S., Böhm C., Keim D., Kriegel H.-P.: *'A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space'*, ACM PODS Symposium on Principles of Database Systems, 1997, Tucson, Arizona.

[6] Berchtold S., Keim D., Kriegel H.-P.: *'The X-Tree: An Index Structure for High-Dimensional Data'*, 22nd Conf. on Very Large Databases, 1996, Bombay, India, pp. 28-39.

[7] Berchtold S., Ertl B., Keim D., Kriegel H.-P., Seidl T.: *'Fast Nearest Neighbor Search in High-Dimensional Spaces'*, Proc. 14th Int. Conf. on Data Engineering, Orlando, 1998.

[8] Beyer K., Goldstein J., Ramakrishnan R., Shaft U..: *'When Is "Nearest Neighbor" Meaningful?'*, submitted for publication, 1998.

[9] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, NJ, 1990, pp. 322-331.

[10] Comer D.: *'The Ubiquitous B-tree'*, ACM Computing Surveys 11(2), 1979, pp. 121-138

[11] Chaudhuri S., Dayal U.: *'Data Warehousing and OLAP for Decision Support'*, Tutorial, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, Tucson, Arizona.

[12] Faloutsos C., Barber R., Flickner M., Hafner J., et al.: *'Efficient and Effective Querying by Image Content'*, Journal of Intelligent Information Systems, 1994, Vol. 3, pp. 231-262.

[13] Faloutsos C., Bhagwat P.: *'Declustering Using Fractals'*, PDIS Journal of Parallel and Distributed Information Systems, 1993, pp. 18-25.

[14] Friedman J. H., Bentley J. L., Finkel R. A.: *'An Algorithm for Finding Best Matches in Logarithmic Expected Time'*, ACM Transactions on Mathematical Software, Vol. 3, No. 3, September 1977, pp. 209-226.

[15] Hjaltason G. R., Samet H.: *'Ranking in Spatial Databases'*, Proc. 4th Int. Symp. on Large Spatial Databases, Portland, ME, 1995, pp. 83-95.

[16] Jagadish H. V.: *'A Retrieval Technique for Similar Shapes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1991, pp. 208-217.

[17] Jain R, White D.A.: *'Similarity Indexing: Algorithms and Performance'*, Proc. SPIE Storage and Retrieval for Image and Video Databases IV, Vol. 2670, San Jose, CA, 1996, pp. 62-75.

[18] Katayama N., Satoh S.: *'The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997, pp. 369-380.

[19] Lin K., Jagadish H. V., Faloutsos C.: *'The TV-Tree: An Index Structure for High-Dimensional Data'*, VLDB Journal, Vol. 3, pp. 517-542, 1995.

[20] Mehrotra R., Gary J.: *'Feature-Based Retrieval of Similar Shapes'*, Proc. 9th Int. Conf. on Data Engeneering, April 1993

[21] Robinson J. T.: *'The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1981, pp. 10-18.

[22] Seidl T., Kriegel H.-P.: *'Efficient User-Adaptable Similarity Search in Large Multimedia Databases'*, Proc. 23rd Int. Conf. on Very Large Databases (VLDB'97), Athens, Greece, 1997.

[23] White D.A., Jain R.: *'Similarity indexing with the SS-tree'*, Proc. 12th Int. Conf on Data Engineering, New Orleans, LA, 1996.