Extending High-Dimensional Indexing Techniques Pyramid and iMinMax(θ) : Lessons Learned

Karthik Ganesan Pillai, Liessman Sturlaugson,
Juan M. Banda, and Rafal A. Angryk

Montana State University, Bozeman, Montana - 59717-3380, USA
{k.ganesanpillai,liessman.sturlaugson,
juan.banda,angryk}@cs.montana.edu

Abstract. Pyramid Technique and iMinMax(θ) are two popular high-dimensional indexing approaches that map points in a high-dimensional space to a single-dimensional index. In this work, we perform the first independent experimental evaluation of Pyramid Technique and iMinMax(θ), and discuss in detail promising extensions for testing k-Nearest Neighbor (kNN) and range queries. For datasets with skewed distributions, the parameters of these algorithms must be tuned to maintain balanced partitions. We show that, by using the medians of the distribution we can optimize these parameters. For the Pyramid Technique, different approximate median methods on data space partitioning are experimentally compared using kNN queries. For the iMinMax(θ), the default parameter setting and parameters tuned using the distribution median are experimentally compared using range queries. Also, as proposed in the iMinMax(θ) paper, we investigated the benefit of maintaining a parameter to account for the skewness of each dimension separately instead of a single parameter over all the dimensions.

Keywords: high-dimensional indexing, iMinMax, Pyramid Technique

1 Introduction

Efficient indexing structures exist for storing and querying low-dimensional data. The B$^+$-tree [2] offers low-cost insert, delete, and search operations for single-dimensional data. The R tree [7] extends the concepts of the B$^+$-tree to 2 or more dimensions by inserting minimum bounding rectangles into the keys of the tree. The R* tree [3] improves the performance of the R tree by reducing the area, margin and overlap of the rectangles. Unfortunately, the performance of these hierarchial index structures deteriorates when employed to handle highly dimensional data [8]. On data with more than 8 dimensions, most of these techniques perform worse than a sequential scan of the data and this performance degradation has come to be called the "curse of dimensionality" [5]. Improved techniques for indexing highly dimensional data are necessary.

One popular approach in addressing the problem of highly dimensional data is to employ an algorithm that maps the values of a high-dimensional record

to a single-dimensional index [10]. After the data is collapsed to this single-dimensional index, it is possible to re-use existing algorithms and data structures that are optimized for handling single-dimensional data, such as the B^+-tree, which offers fast insert, update, and delete operations. An advantage of mapping to a single dimension and using a B^+-tree is that the algorithms can be easily implemented on top of an existing DBMS [4]. The most widely cited examples of this strategy include the Pyramid Technique [4], and the iMinMax(θ) [8].

Both the Pyramid Technique (PT) and iMinMax(θ) partition the data space into different partitions that map the high-dimensional data to a single dimensional value. To partition the data space, the PT uses the center point of the data space [4], and the iMinMax(θ) uses the dimension that has the largest or smallest value [8]. For data sets with skewed distributions, these two indexing techniques can result in a disproportionate number of points in the resulting partitions. In this paper, we show that by using medians (approximate and actual) of the skewed distribution in the data we can improve the partitioning strategy in a way that will better balance the number of points in resulting partitions.

The rest of the paper is organized as follows: In section 2, we give a brief background of PT and iMinMax(θ). In section 3 we explain PT and our proposed extensions in detail. In section 4 we explain iMinMax(θ) and our proposed extensions in detail. Finally, we present a large variety of experiments demonstrating the impact of extensions to PT and iMinMax(θ), and conclude with a summary of results.

2 Brief Background

The PT introduced in [4], partitions a data space of d dimensions into $2d$ hyper-pyramids, with the top of each pyramid meeting at the center of the data space. The index of each point has an integer part and a decimal part. The integer part refers to the pyramid in which the point can be found, and the decimal part refers to the "height" of the point within that pyramid. Although two points may be relatively far apart in the data space, any number of points can potentially be mapped to the same index, because each point is indexed by a single real value.

Following a similar strategy, the iMinMax(θ), first presented in [8], maps each point to the "closest edge/limit" of the data space instead of explicitly partitioning the data space into pyramids. By mapping to axes instead of pyramids, they reduce the number of partitions from $2d$ to d. The simple mapping function was also intended to avoid more costly pyramid intersection calculations.

Mapping to a single dimension from multiple dimensions results in a lossy transformation. Therefore, both of these techniques must employ a filter and refine strategy. To be useful, the transformation should allow much of the data to be ignored for a given query. At the same time, the transformation must ensure that the filter step misses no true positives, so that, after the refine step removes the false positives, the result is *exactly* the points that match the query.

3 The Pyramid Technique

The PT [4] divides the high-dimensional data space to $2d$ pyramids (see Fig. 1), each of which share the center point of the data space as the top of the pyramid

and have a $(d-1)$-dimensional surface of the data space as the base of the pyramid. A point in the data space is associated to a unique pyramid. The point also has a height within its pyramid. This association of point to a pyramid is called the pyramid number of a point in the data space. Any order-preserving one-dimensional index structure can be used to index pyramid values. For both insert and range query processing, the d-dimensional input is transformed into a 1-dimensional value, which can be processed by the B^+-tree. The leaf nodes of the B^+-tree store the d-dimensional point value and the 1-dimensional key value. Thus, inverse transformation is not necessary. In the next section, the data space partitioning will be explained in greater detail.

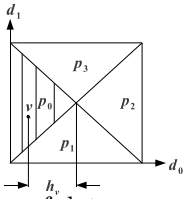

Fig. 1: Partition of data space into pyramids

3.1 Index Creation

The partition of the data space in the PT follows two steps. In the first step, the d-dimensional data space is divided into $2d$ pyramids, with the center point of the data space as the top of each pyramid and a $(d-1)$-dimensional surface as the base of each pyramid. In the second step, each pyramid is divided into multiple partitions, each partition corresponding to one data page of the B^+-tree. Fig. 1 shows the partitioning of a 2-dimensional data space into four pyramids p_0, p_1, p_2, and p_3, which all have the center point of the data space as the top and one edge of the data space as the base. Also the partitions within pyramid p_0, and height (h_v) of a point v in pyramid p_0 are shown in the figure.

The pyramid value of a point (pv_v), is the sum of the pyramid number and the height of the point within that pyramid. Calculation of the pyramid number and the height of a point is shown in *Algorithm* 1. In this algorithm, D is the total number of dimensions, d_{max} is the pyramid number in the data space partition. The algorithm assumes the data space has been normalized so that the center of the data space is at 0.5 in each dimension. Using the pv_v as a key, the d-dimensional point is inserted in the B^+-tree in the corresponding data page of the B^+-tree.

3.2 Query Processing

We now discuss point queries, range queries, and kNN queries in this section. The general definition of **point query**, can be stated as follows "Given a query point q, decide whether q is in the database." This problem can be solved by first finding the pyramid value pv_q of the query point q and querying the B^+-tree using pv_q. Thus, d-dimensional results are obtained sharing the pyramid value pv_q. From this result, we scan and determine whether the result contains q and output the result.

Algorithm 1 To calculate the pyramid value of a point v, adapted from [4]

```
PyramidValue(Point v)
d_max = 0
height = |0.5 − v[0]|
for j = 1 → D − 1 do
   if height < |0.5 − v[j]| then
      d_max = j
      height = |0.5 − v[j]|
   end if
end for
if v[d_max] < 0.5 then
   i = d_max
else
   i = d_max + D
end if
pv_v = i + height
return pv_v
```

Second, for **range queries**, which are stated as "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." Range query processing using the PT is a complex operation, a query rectangle of a range query might intersect several pyramids, and computation of the area of the interval is not trivial.

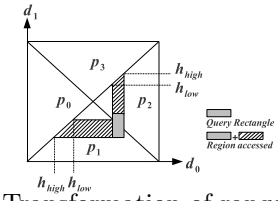

Fig. 2: Transformation of range queries

Fig. 2 shows a query rectangle and the region accessed for that query rectangle. This computation of the area is a two-step process. First, we need to determine which pyramids intersect with the query rectangle, and second, we need to determine the height intervals inside the pyramids. To determine the interval inside a pyramid (h_v between two values) for all objects is a one-dimensional indexing problem. Next, a simple point-in-rectangle test is performed in the refinement step.

An algorithm to find pyramid intersections and the interval within each pyramid for range queries is given in *Algorithm* 2, and it uses Equations 1 through 8. In this algorithm, the given query rectangle is first transformed in a way that the interval is defined relative to the center point. Next, pyramids in the data space partition that intersect with the query rectangle are found and the interval $[i + h_{low}, i + h_{high}]$ inside each intersected pyramid is computed. Using this interval, a point-in-rectangle test is performed using the B^+-tree.

$$intersect = \begin{cases} true & \text{if } k = D \\ false & \text{if } k \neq D \end{cases} \tag{1}$$

where k is obtained from equation 2

Algorithm 2 To process range query $[\hat{qr}_{0_{min}}, \hat{qr}_{0_{max}}], ..., [\hat{qr}_{d-1_{min}}, \hat{qr}_{d-1_{max}}]$, adapted from [4]

RangeQuery($qr[D][2]$)
// Initialize variables
$sq[D][2]$
$qw[D][2]$
for $i = 0 \rightarrow D - 1$ **do**
  $sq[i][0] = qr[i][0] - 0.5$
  $sq[i][1] = qr[i][1] - 0.5$
end for
for $i = 0 \rightarrow (2D) - 1$ **do**
  if $(i < D)$ **then**
    $qw[i][0] = sq[i][0]$
    $qw[i][1] = sq[i][1]$
  end if
  if $(i \geq D)$ **then**
    $qw[i - D][0] = sq[i - D][0]$
    $qw[i - D][1] = sq[i - D][1]$
  end if
  // Using Equation 1
  if intersect **then**
    if $(i < D) \wedge (qw[i][1] > 0)$ **then**
      $qw[i][1] = 0$
    end if
    if $(i \geq D) \wedge (qw[i - D][0] < 0)$ **then**
      $qw[i - D][0] = 0$
    end if
    // Using Equation 6 and 8
    Find h_{low} and h_{high}
    Search B^+-tree
  end if
end for

$$k = \begin{cases} \forall j, 0 \leq j < D, \hat{q}_{i_{min}} \leq -MIN(\hat{q}_j) : & \text{if } i < D \\ \forall j, 0 \leq j < D, \hat{q}_{i-D_{min}} \geq -MIN(\hat{q}_j) : & \text{if } D - 1 < i \end{cases} \tag{2}$$

$$\bar{q}_j = \begin{cases} MIN(q_j) & \text{if MIN}(q_j) > \text{MIN}(q_i) \\ MIN(q_i) & otherwise \end{cases} \tag{3}$$

$$MIN(r) = \begin{cases} 0 & \text{if } r_{min} \leq 0 \leq r_{max} \\ min(|r_{min}|, |r_{max}|) & otherwise \end{cases} \tag{4}$$

$$MAX(r) = max(|r_{min}|, |r_{max}|) \tag{5}$$

$$h_{high} = MAX(\hat{q}_i) \tag{6}$$

$$hvalue = max_{0 \leq j < D} : (\bar{q}_j)(*) \tag{7}$$

$$h_{low} = \begin{cases} 0 & \text{if } \forall j, 0 \leq j < D : (\hat{q}_{j_{min}} \leq 0 \leq \hat{q}_{j_{max}}) \\ hvalue & otherwise \end{cases} \tag{8}$$

Finally, we address ***k*NN** queries, which are stated as "Given a set S of n d-dimensional data points and a query point q, the kNN search is to find subset $S' \subseteq S$ of $k \leq n$ data points such that for any data point $u \in S'$ and $v \in S - S'$, $dist(u, q) \leq dist(v, q)$". The procedure to perform a kNN search using the decreasing-radius kNN search technique, introduced in [9], is given in *Algorithm* 3. In this method, after finding the pyramid number for the given query point q, the B^+-tree is searched to locate the leaf node that has the key value for the qiven point q, or the largest key value less than the key value of q. Once the key value is identified, the function $SEARCHLEFT$ ($SEARCHRIGHT$) is used to check the data points of the node towards the left (right) to determine if they are among the k nearest neighbors. When the difference between the current key value in the node and the pyramid value of q is greater than D_{max} and there are k data points in A or the key value of the leaf node is less (greater) than i $(i + 0.5)$, the search on left (right) stops. In the next step, a query square W is generated to perform an orthogonal range search. The rest of the pyramids are examined one by one, and if a pyramid intersects with W, a $RangeQuery(W)$ is performed to check if the data points in this pyramid intersecting W are among the k nearest neighbors. The side length of W is updated after each pyramid is examined. The algorithm stops once all the pyramids have been examined.

Algorithm 3 The decreasing radius Pyramid kNN search algorithm, adapted from [9]

```
Pyramidk NN(Point q, int k)
A ← emptyset
i ← pyramid number of the pyramid q is in
node ← LocateLeft(T, q)
SearchLeft(node, A, q, i)
SearchRight(node, A, q, i + 0.5)
D_max ← D(A_0, q)
Generate W centered at q with △ ← 2D_max
for j = 0 → 2D − 1 do
   if (j ≠ i) ∧ (W intersects pyramid j) then
      RangeQuery(W)
      Update W with updated △ ← 2D_max
   end if
end for
return A
```

3.3 Extending Pyramid Technique

The data partitioning strategy of the original PT assumes a uniform data distribution. For clustered data, as shown in Fig. 3 (a), most of the data is contained in only a few pyramids. Partitioning this data space will result in sub-optimal space partition, as shown in Fig. 3 (b). A better partitioning approach is shown in Fig. 3 (c).

In the Extended PT, the basic idea is to let the pyramid's center point to follow the center of the actual data distribution. Thus, the data space is mapped to the canonical data space $[0, 1]^d$ such that the d-dimensional median of the data space is mapped to the center point. The transformation is applied only

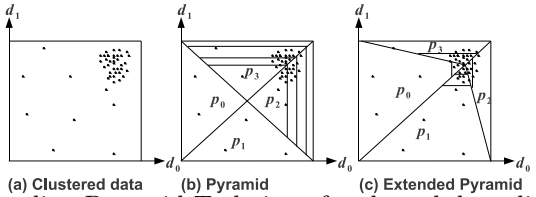

Fig. 3: Extending Pyramid Technique for skewed data distributions

to determine the pyramid value of points and query rectangles, and hence an inverse transformation is again unnecessary.

Computing the d-dimensional median is a hard problem [4] and thus two different approaches for finding the approximate median are explored. In the first approach, a histogram is created for each dimension. From the created histogram, the bin containing the median is found, and then that bin is searched for the median. This method requires n comparisons. In the second approach, we use the approximate median finding algorithm described in [1]. This method requires fewer than $(4/3) \times n$ comparisons and $(1/3) \times n$ exchanges on average, and fewer than $(3/2) \times n$ comparisons, with $(1/2) \times n$ exchanges in the worst case. The algorithm takes $n = 3^r$ as the input where r is an integer and proceeds in r stages. At each stage the algorithm divides the input into subsets of three elements, and calculates the median of each such triplet. Medians of each triplet are used in the next stage. The algorithm continues recursively, using the medians saved from the previous stage to compute the approximate median of the initial set. The method described can be generalized to array sizes which are not powers of three as follows: Let the array size be n, where $n = 3 \times t + k$, and where $k \in 0, 1, 2$. Then the first t triplets have their median extracted as before, and the t selected medians, as well as the remaining k elements are forwarded to the next stage. Moreover the direction in which those first triplets are selected alternates—either left-to-right or right-to-left. This ensures that no elements remain for more than one iteration. The algorithm continues iteratively using the results of each stage as input for the next one. This is done until the number of elements falls below a small fixed threshold. Finally, the elements are sorted to obtain their median as the final result.

The computation of the median can be done dynamically, in the case of dynamic insertions, or once in the case of a bulk load of points. This d-dimensional approximate median may lie outside the convex hull of the data cluster. Given the d-dimensional median mp_i of the data set, a set of d functions t_i are defined in [4], $0 \leq i < (d-1)$ transforming the given data space in dimension i such that the transformed data space still has a range of $[0, 1]^d$, that the median of the data becomes the center point of the data space, and that each point in the original data space is mapped to a point inside the canonical data space.

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

To insert a point v into an index, transform v into a point such that $v'_i = t_i(v_i)$ and determine the pyramid value $pv_{v'}$. Using $pv_{v'}$, point v is inserted into the B^+-tree. To process a query, first transform the query rectangle q into a query rectangle q' such that $q'_{i_{min}} = t_i(q_{i_{min}})$ and $q'_{i_{max}} = t_i(q_{i_{max}})$. Next, algorithms discussed in earlier sections are used to determine intersection of pyramids and

$$q_j = \begin{cases} [j + \max_{i=1}^{d} x_{il}, j + x_{jh}] & \text{if } \min_{i=1}^{d} x_{il} + \theta \geq 1 - \max_{i=1}^{d} x_{il} \\ [j + x_{jl}, j + \min_{i=1}^{d} x_{ih}] & \text{if } \min_{i=1}^{d} x_{ih} + \theta < 1 - \max_{i=1}^{d} x_{ih} \\ [j + x_{jl}, j + x_{jh}] & \text{otherwise} \end{cases} \quad (9)$$

ranges within pyramids to find the points in the query rectangle. Finally, the refine step is performed to filter out false positives as before.

4 The iMinMax(θ) Algorithm

The iMinMax(θ) algorithm maps a d-dimensional space to a one-dimensional space, by indexing on the "edges". The maximum or minimum value among all the dimensions of a point is called an "edge" [8]. The iMinMax(θ) technique uses either the *Max edge* or *Min edge* in the index keys for the points.

4.1 Index Creation

As with the PT, the data is normalized such that each data point x resides in a unit d-dimensional space. A data point x is denoted as $x = (x_1, x_2, \ldots, x_d)$ where $x_i \in [0, 1]\ \forall i$. Let $x_{max} = \max_{i=1}^{d} x_i$ and $x_{min} = \min_{i=1}^{d} x_i$. Each point is mapped to a single dimensional index value $f(x)$ as follows:

$$f(x) = \begin{cases} d_{min} + x_{min}, & \text{if } x_{min} + \theta < 1 - x_{max} \\ d_{max} + x_{max}, & \text{otherwise} \end{cases}$$

The parameter θ can be tuned to account for skewed data distributions in which much of the data would otherwise be mapped to the same edge, resulting in a less efficient search through the B^+-tree. In the simple case when $\theta = 0$, each point is mapped to the axis of the closest edge which is appropriate for uniformly distributed data. When $\theta > 0$, the mapping function is biased toward the axis of the maximum value, while $\theta < 0$ biases it toward the axis of the minimum value.

4.2 Query Processing

Range queries are first transformed into d one-dimensional subqueries. The range query interval for the jth dimension, denoted q_j, is calculated by Equation 9. The variables x_{il} and x_{ih} represent the low and high bound, respectively, for the range interval in the ith dimension. In the original iMinMax paper [8], they prove that the union of the results from the d subqueries is guaranteed to return the set of all points found within the range, while no smaller interval on the subqueries can guarantee this. Moreover, they prove that at most d subqueries must be performed. In fact, they prove that a subquery $q_i = [l_i, h_i]$ need not be evaluated if one of the following holds:

$$\min_{j=1}^{d} x_{jl} + \theta \geq 1 - \max_{j=1}^{d} x_{jl} \quad \text{and} \quad h_i < \max_{j=1}^{d} x_{jl}$$

$$\min_{j=1}^{d} x_{jh} + \theta < 1 - \max_{j=1}^{d} x_{jh} \quad \text{and} \quad l_i > \min_{j=1}^{d} x_{jh}$$

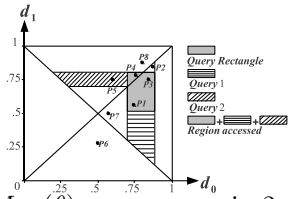

Fig. 4: Example iMinMax(θ) range query in 2 dimensions with $\theta = 0$

This occurs when all the answers for a given subquery are along either the *Max* or the *Min* edge. Thus, if either of these conditions hold, the answer set for q_i is guaranteed to be empty and can be ignored.

Fig. 4 shows the two subqueries generated by an example range query in 2 dimensions when $\theta = 0$. Query 1 returns $\{P1, P2, P3\}$ during the filter step and then refines by removing $P2$. Likewise, Query 2 returns $\{P4, P5\}$ and then refines by removing $P5$.

4.3 Tuning the θ Parameter

A d-dimensional median point can be used to calculate the optimal θ, denoted θ_{opt}. This d-dimensional median is calculated by first finding the median for each dimension. The combination of these d one-dimensional medians forms the median point used to calculate θ_{opt}. The x_{min} and x_{max} of this median are then used to calculate the optimal θ as

$$\theta_{opt} = 1 - x_{max} - x_{min}$$

4.4 Extending iMinMax(θ) with Multiple θ's

In this section, we investigate an extension to the iMinMax(θ) algorithm that incorporates multiple θ's, instead of using only the single θ parameter. For this extension, each dimension i will have its own unique parameter θ_i. The original θ parameter attempts to create a balanced partition of the data, making the median of the dataset a good choice from which to compute the θ parameter. This median, however, is calculated by finding the median of each dimensions individually. Instead of combining these medians into a single median and computing a single θ parameter, each of these medians is now used individually to compute each θ_i parameter.

This extension changes the mapping function when computing the single-dimension index value of a point. Notice that the mapping function compares only two dimensions at a time, the dimensions of the minimum and maximum values. The multiple θ_i parameters account for potentially different skewness across *pairs* of dimensions. Let θ_{min} and θ_{max} be the θ_i parameters for the dimensions of the minimum and maximum values, respectively. The mapping function now becomes:

$$f(x) = \begin{cases} d_{min} + x_{min}, & \text{if } x_{min} + \theta_{min} < 1 - x_{max} - \theta_{max} \\ d_{max} + x_{max}, & \text{otherwise} \end{cases}$$

The mapping of the range subqueries and the criteria for subquery pruning are modified similarly. The introduction of multiple θ_i parameters does not add significant overhead to index creation, with the number of θ_i parameters scaling linearly with the number of dimensions and with the mapping function only requiring one additional term.

5 Experiments

In this section we first present the experiments and results of PT and proposed extensions, followed by the experiments and results of iMinMax(θ) and proposed extensions, respectively. The benefits of using medians and the influence of different approximate median methods is discussed for PT and the effects of tuning the θ parameter and multiple θ's are discussed for iMinMax(θ).

5.1 Pyramid Technique (PT)

For the PT, we performed two different experiments on skewed distributions. For the first experiment, we measured the influence of approximate median methods on tree construction time and kNN query performance across different dimensions. For this experiment a total of three data sets are created with $4, 8$, and 16 dimensions and each data set has $500,000$ data points. For the second experiment, we measured the influence of approximate median methods on kNN query performance across different data set size. For this experiment a total of three data sets are created with $500,000$, $750,000$, and $1,000,000$ data points and each data set has 16 dimensions.

The results (see Figs. 5, 6, and 7) for the PT show how different approximate median methods influence tree construction time and kNN query performance. Each of the figures for PT shows four lines representing the various methods (see Table 1).

Table 1: Summary of experiments on Pyramid Indexing technique (Names of experiment are used also as legends in Figs. 5, 6, 7)

Name	Experiment Description
PT	Pyramid Technique
HMPT	Histogram-based approximate median PT
AMPT	Approximate median PT
TMPT	Brute force median PT

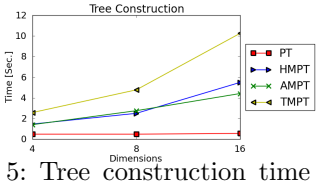

Fig. 5: Tree construction time over dimensionality of data space

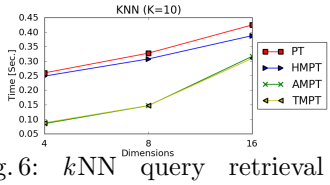

Fig. 6: kNN query retrieval behaviour over data space dimension

First, the benefit of using medians in PT and the influence of different approximate median methods is shown in Fig. 6 and 7 for kNN query on single

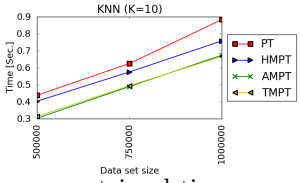

Fig. 7: kNN query retrieval time over database size

clustered data sets with different dimensions and on single clustered data sets with different sizes. Each datapoint on the two graphs represents query retrieval time averaged over $1,000$ kNN ($k = 10$) random queries. The centroid of the cluster is offset from the center of the space, while the original PT does not account for this skewed distribution in the data. On the other hand, the other methods compute the d-dimensional (approximate) median to be the center of the data space.

This results in better space partitioning for the single-cluster datasets, because data points in the data space are more evenly mapped to different pyramids (see Fig. 3 (c)), thus improving the performance of the kNN queries. Moreover, from the figures we can observe that AMPT performance is close to TMPT in comparison to HMPT. AMPT, with its high probability of finding an approximate median within a very small neighborhood of the true median [1], leads to better space partitioning, hence results in better performance. These results demonstrate the benefit of using approximate median methods with PT on a single skewed distribution.

However, computing the exact or approximate median increases the time to build the index and we can observe this from Fig. 5. In order to reduce the time to build index for extended PT, we should use an approximate median method that is computationally less expensive.

5.2 iMinMax(θ)

For the iMinMax(θ), we performed two different experiments. For the first experiment, we measured the influence of setting a θ value on range query performance across different dimensions. For this experiment a total of three skewed distribution data sets are created with $4, 8$, and 16 dimensions and each data set has $500,000$ data points with single cluster. For the second experiment, we measured the influence of calculating a unique θ_i for each dimension i on range query performance across different dimensions. For this experiment a total of three data sets are created with $4, 8$, and 16 dimensions and each data set has $500,000$ data points. Moreover, for this experiment all the data set have ten clusters and each cluster has $50,000$ data points. To measure the performance of range queries, two different range queries are generated. For the narrow range query we picked $1,000$ random query points from the data set and modified a single dimension value of each point with +/- 0.01, and for the wide range query we selected $1,000$ random query points from the data set and modified half of the dimensions value of each point with +/- 0.01.

The results (see Figs. 8 through 13) for the iMinMax(θ) show how different values for the θ parameter (see Table 2) influence range query performance. The

Table 2: Summary of experiments on iMinMax(θ) Indexing technique (Names of experiment are used also as legends in Figs. 8 through 13)

Name	Experiment Description
DEF	Default setting when $\theta = 0$, which assumes that the center of the space is the centroid of the data
OPT	Single θ calculated by finding the exact median of the dataset
APP	Single θ calculated by finding the approximate median
MOPT	Multiple θ's calculated by finding the exact median for each dimension separately
MAPP	Multiple θ's calculated by finding the approximate median for each dimension separately

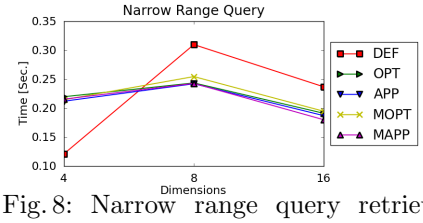

Fig. 8: Narrow range query retrieval time on tuning θ for single cluster

Fig. 9: Wide range query retrieval time on tuning θ for single cluster

results show that, by setting θ as calculated using either the exact or approximate median, the performance improves upon the default $\theta = 0$ as the number of dimensions increases.

Effects of tuning the θ parameter The benefit of tuning θ is shown in Fig. 8 and 9 for the narrow range query and wide range query with the single-cluster dataset. Each datapoint on the two graphs represents the query time averaged over $1,000$ range queries each centered at random points drawn from the dataset. The centroid of the cluster is offset from the center of the space, while the default $\theta = 0$ does not account for this skewed distribution in the data. On the other hand, the other methods that calculate either one or multiple θ's are able to account for the skewed distribution and and are able to better optimize the range queries as the number of dimensions increases. With this dataset, the medians for the different dimensions do not vary significantly, and thus the difference between the single and multiple θ's is minimal.

Effects of multiple θ_i's The benefit of calculating a unique θ_i for each dimension i is shown in Figs. 10 and 11 for the narrow range query and for the wide range query with the ten-cluster dataset. Again, each datapoint on the two graphs represents the query time averaged over $1,000$ range queries each centered at random points drawn from the dataset. For this dataset, the medians of different dimensions vary by a difference of up to 41% of the space. This time the differences between the single θ and multiple θ's become more apparent. By maintaining a θ_i for each dimension, the *MOPT* and *MAPP* methods perform as well or better than the *OPT* and *APP* methods using the single θ.

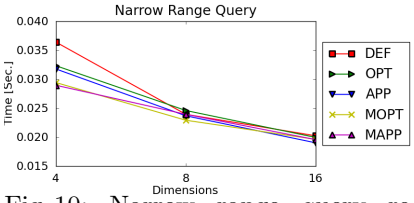

Fig. 10: Narrow range query retrieval time on calculating a unique θ_i for ten cluster

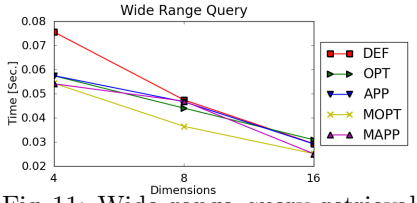

Fig. 11: Wide range query retrieval time on calculating a unique θ_i for ten cluster

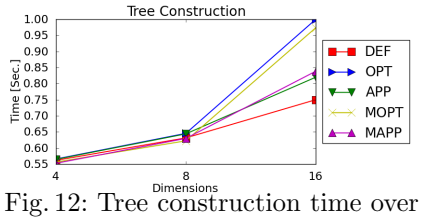

Fig. 12: Tree construction time over dimensionality of data space for single cluster

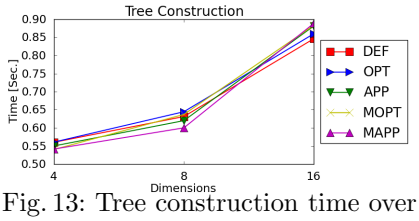

Fig. 13: Tree construction time over dimensionality of data space for ten cluster

On the other hand, calculating the exact or approximate median does not add significant time complexity to the creation and population of the B$^+$-tree, as shown by Figs. 12 and 13 for the single-cluster and ten-cluster datasets. These results demonstrate the benefit of calculating θ from the median of the dataset with a skewed distribution and the benefit of calculating θ_i from the median of each dimension i of the dataset when the magnitude of the skew varies across different dimensions.

6 Conclusions

From the experiments with the PT (Figs. 6 and 7), it is demonstrated that using the d-dimensional median to be the center of the data space and mapping the given skewed data set to the canonical data space $[0, 1]^d$ results in better space partitioning, thus improving the performance of query retrieval time. Furthermore the influence of computational complexity (Fig. 5) and closeness to median of approximate median methods is demonstrated from the experiments.

From the experiments with the iMinMax(θ) algorithm (Figs. 8 and 9), it is demonstrated that deriving the θ parameter from the median offers improved range query performance over the default parameter setting for skewed datasets. Furthermore, calculating a unique θ_i for each dimension i (Figs. 10 and 11) can improve the performance of range queries for datasets with skewness that varies across different dimensions with little extra computational effort.

In this paper, we have shown experimentally for both PT and iMinMax(θ) algorithm, that by using the median (approximate) of the skewed distributions in the data, we can partition the data space into different partitions with proportionate number of points in each partition. Effectiveness is expected to increase as the dimensionality and data volume increases for skewed data distributions. In future works, we plan to evaluate the performance of PT and iMinMax(θ)

with their extensions on larger data sets, and more dimensions. We also, plan to compare our proposed extensions of PT and iMinMax(θ) algorithms with the approach described by Günnemann *et al.* in [6], as they also address indexing high-dimensional data that have skewed distributions.

7 Acknowledgments

This work was supported by two National Aeronautics and Space Administration (NASA) grant awards, 1) No. NNX09AB03G and 2) No. NNX11AM13A.

References

1. S. Battiato, D. Cantone, D. Catalano, G. Cincotti, and M. Hofri. An efficient algorithm for the approximate median selection problem. In G. Bongiovanni, R. Petreschi, and G. Gambosi, editors, *Algorithms and Complexity*, volume 1767 of *Lecture Notes in Computer Science*, pages 226–238. Springer Berlin / Heidelberg, 2000.
2. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972. 10.1007/BF00288683.
3. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19:322–331, May 1990.
4. S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. *SIGMOD Rec.*, 27:142–153, June 1998.
5. S. Berchtold and D. A. Keim. High-dimensional index structures database support for next decade's applications. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 501–, New York, NY, USA, 1998. ACM.
6. S. Günnemann, H. Kremer, D. Lenhard, and T. Seidl. Subspace clustering for indexing high dimensional data: a main memory index based on local reductions and individual multi-representations. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 237–248, New York, NY, USA, 2011. ACM.
7. A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, June 1984.
8. 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 *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '00, pages 166–174, New York, NY, USA, 2000. ACM.
9. Q. Shi and B. Nickerson. Decreasing Radius K-Nearest Neighbor Search Using Mapping-based Indexing Schemes. Technical report, University of New Brunswick, 2006.
10. C. Yu, S. Bressan, B. C. Ooi, and K.-L. Tan. Querying high-dimensional data in single-dimensional space. *The VLDB Journal*, 13:105–119, May 2004.