The VLDB Journal (2002) / Digital Object Identifier (DOI) 10.1007/s00778-004-0121-9

Querying high-dimensional data in single-dimensional space

Cui Yu[1], **Stéphane Bressan**[2], **Beng Chin Ooi**[2], **Kian-Lee Tan**[2]

[1] Department of Computer Science, Monmouth University, West Long Branch, NJ 07764, USA
e-mail: cyu@monmouth.edu
[2] Department of Computer Science, National University of Singapore, 3 Science Drive 2, Singapore 117543
e-mail: {steph, tankl}@nus.edu.sg, ooibc@comp.nus.edu.sg

Edited by ♣. Received: ♣ / Revised version: ♣
Published online: ♣ 2004 –

Abstract. In this paper, we propose a new tunable index scheme, called iMinMax(θ), that maps points in high-dimensional spaces to single-dimensional values determined by their maximum or minimum values among all dimensions. By varying the tuning "knob", θ, we can obtain different families of iMinMax structures that are optimized for different distributions of data sets. The transformed data can then be indexed using existing single-dimensional indexing structures such as the B^+-trees. Queries in the high-dimensional space have to be transformed into queries in the single-dimensional space and evaluated there. We present efficient algorithms for evaluating window queries as range queries on the single-dimensional space. We conducted an extensive performance study to evaluate the effectiveness of the proposed schemes. Our results show that iMinMax(θ) outperforms existing techniques, including the Pyramid scheme and VA-file, by a wide margin. We then describe how iMinMax could be used in approximate K-nearest neighbor (KNN) search, and we present a comparative study against the recently proposed iDistance, a specialized KNN indexing method.

Keywords: High-dimensional data – Single-dimensional space – Window and KNN queries – Edge – iMinMax(θ)

1 Introduction

With an increasing number of new database applications such as multimedia-content-based retrieval and time series matching that deal with high-dimensional databases, the design of efficient indexing and query processing techniques over high-dimensional data sets becomes an important research area. Typical operations of high-dimensional applications include window-based query retrieval, similarity range, and K-nearest neighbor (KNN) searches. While similarity range and KNN searches are frequently used in high-dimensional databases for finding objects with similar properties, window query search is required to sieve out interesting data for further analysis. For example, in a scientific database application, the physicist searches for interesting events of high-energy physics experiments by specifying ranges over the attributes or similar events based on a known result. Therefore, it is important for high-dimensional databases to have indexes supporting both window-based and similarity range and KNN searches. In this paper, we shall concentrate on the indexing problem of high-dimensional databases for window search while still supporting fast approximate KNN search.

Many multidimensional indexing structures have been proposed in the literature [4,10]. However, it has been observed that the performance of hierarchical tree index structures such as R-trees [8] and R^*-trees [1] deteriorates rapidly with the increase in the dimensionality of data. This phenomenon can be explained as follows. As the number of dimensions increases, the area covered by the query increases tremendously. Consider a hypercube with a selectivity of 0.1% of the domain space ([0,1],[0,1],. . .,[0,1]). This is a relatively small query in two- to three-dimensional databases. However, for a 40-dimensional space, the query width along each dimension works out to be 0.841, which causes the query to cover a large area of the domain space. Consequently, many leaf nodes of a hierarchical index have to be searched. The degradation in performance can be so severe that a simple sequential scan of the index keys becomes the preferred method [2,15]. However, sequential scanning is expensive as it requires the whole database to be searched for any range queries, irrespective of query sizes. Therefore, research efforts have aimed at developing techniques that can outperform sequential scanning. Some of the notable techniques include the VA-file [15] and the Pyramid scheme [2].

In this paper, we adopt a tranformational approach to reduce high-dimensional data to a single dimensional value. Our scheme is motivated by two observations. First, data points in high-dimensional space can be ordered based on the maximum value of all dimensions.[1] Second, it is possible to determine a range of values such that data points whose index keys are outside the range will not be in the answer set. The former implies that we can represent high-dimensional data in single-dimensional space and reuse existing single-dimensional indexes. The latter provides a mechanism to prune the search space.

[1] Note that we have adopted the maximum value in our discussion; similar observations can be made with the minimum value.

Our proposed new tunable indexing scheme, iMinMax(θ), has several nice features. First, iMinMax(θ) adopts a simple transformation function to map high-dimensional points to a single-dimensional space. Let x_{min} and x_{max} be, respectively, the smallest and largest values among all the d dimensions of the data point $(x_1, x_2, \ldots, x_d)$ $0 \leq x_j \leq 1, 1 \leq j \leq d$. Let the corresponding dimensions for x_{min} and x_{max} be d_{min} and d_{max}, respectively. The data point is mapped to y over a single-dimensional space as follows:

$$y = \begin{cases} d_{min} \times c + x_{min} & \text{if } x_{min} + \theta < 1 - x_{max} \\ d_{max} \times c + x_{max} & \text{otherwise,} \end{cases}$$

where c is a positive constant to stretch the range. We note that the transformation actually partitions the data space into different partitions based on the dimension that has the largest value or smallest value and provides an ordering within each partition. Second, the B$^+$-tree is used to index the transformed values. Thus, iMinMax(θ) can be implemented on existing DBMSs without additional complexity, making it a practical approach.

Third, in iMinMax(θ), queries on the original space need to be transformed to queries on the transformed space. For a given window query, the range of each dimension is used to generate a range subquery on the dimension. The union of the answers from all subqueries provides the candidate answer set from which the query answers can be obtained. iMinMax(θ)'s query mapping function facilitates effective range query processing: (i) the search space on the transformed space contains all answers from the original query, and it cannot be further constrained without the risk of missing some answers; (ii) the number of points within a search space is reduced; and (iii) some of the subqueries can be pruned away without being evaluated.

Fourth, by varying θ, we can obtain different families of iMinMax(θ) structures. At the extremes, iMinMax(θ) maps all high-dimensional points to the maximum (minimum) value among all dimensions; alternatively, it can be tuned to map some points to the maximum values and others to the minimum values. Thus, iMinMax(θ) can be optimized for data sets with different distributions.

Finally, the iMinMax can be used for approximate nearest-neighbor searches. The index facilitates fast initial response time by providing users with approximate answers that are progressively refined till more accurate answers are obtained. Experiments conducted reveal that the accuracy obtained by the proposed scheme is very close to that obtained in an optimal approach. This is significant as most multidimensional indexes proposed for range and spatial queries are not designed to efficiently support high-dimensional similarity (range and nearest-neighbor) queries, and not all high-dimensional metric-based indexes proposed for similarity queries can be used to support range queries.

We implemented the iMinMax(θ) and evaluated its performance against the more complex Pyramid technique and the VA-file. We conducted an extensive performance study on both uniform and skewed data sets. Our results show that the proposed scheme is more efficient than both the Pyramid technique and the VA-file. We also conducted experiments on the accuracy and performance of the iMinMax in processing approximate KNN searches. The results show that the proposed method is able to return 90% of the actual answers very quickly. Moreover, a comparative study against iDistance[17] shows that iMinMax is a competitive structure for KNN searches.

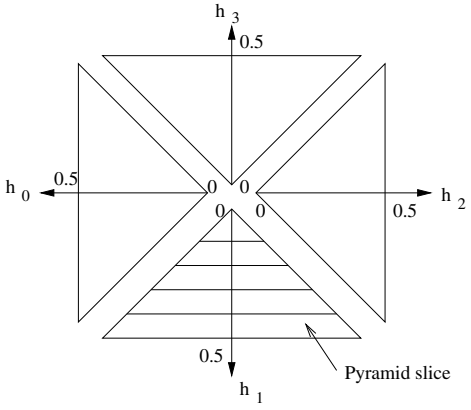


Fig. 1. Index key assignment in the Pyramid technique

A preliminary version of this paper appeared in [12]. There, we presented the basic idea of iMinMax(θ). In this paper, we make the following additional contributions. First, we provide more detailed description of the iMinMax. Second, we conducted a more comprehensive set of experiments to demonstrate the effectiveness of iMinMax(θ). Third, we extended the iMinMax for approximate KNN search and conducted experiments on KNN accuracy and performance.

The rest of this paper is organized as follows. In the next section, we describe related work. In Sect. 3, we present the proposed iMinMax(θ) strategy. In Sect. 4, we present the operations on iMinMax(θ), including point and range search and update operations. Section 5 reports our experimental study and findings, and Sect. 6 describes the extension of iMinMax(θ) to KNN searches and its performance study. Finally, we conclude in Sect. 7 with directions for future work.

2 Related work

There is a considerable body of literature on high-dimensional indexing [6]. In this section we shall review three recent indexing structures that are related to our work and used in our comparison studies, namely, the Pyramid technique [2], the VA-file [15], and the iDistance [17].

The basic idea of the Pyramid technique is to transform the d-dimensional data points into one-dimensional values and then store and access the values using a conventional index such as the B$^+$-tree. It splits the data space into $2d$ pyramids that share the center point of the data space as their top and have $(d-1)$-dimensional surface of the data space as their base. All points located on the i-th $(d-1)$-dimensional surface (the base of the pyramid) have a common property: either their i-th coordinate is 0 or their $(i-d)$-th coordinate is 1. On the other hand, all points ν located in the i-th pyramid p_i have the furthest distance from the top in the i-th dimension. The dimension in which the point has the longest distance from the top determines which pyramid the point lies in.

Another property of the Pyramid technique is that the location of a point ν within its pyramid is indicated by a single value, which is the distance from the point to the center point

according to dimension j_{max} (Fig. 1). The data on the same slice in a pyramid have the same pyramid value. That is, any objects falling on the slice will be represented by the same pyramid value. As a result, many points will be indexed by the same key in a skewed distribution. It has been suggested [2] that the center point can be shifted to handle data skewness. However, this incurs recalculation of all index values, i.e., redistribution of the points among the pyramids, and reconstruction of the B$^+$-tree. To retrieve a point q, the pyramid value P_ν of q is determined and used to search the B$^+$-tree. All points with P_ν will be checked and retrieved. To perform a range query, the pyramids that intersect the search region are first obtained, and for each pyramid a subquery range is worked out. Each subquery is then used to search the B$^+$-tree. For each range query, $2d$ subqueries may be required, one against each pyramid. For window queries, the Pyramid technique has been shown to be more efficient than the X-tree [3] and the Hilbert-R-tree [9]. The iMinMax(θ) differs from the Pyramid technique in the following ways: (1) the distribution of data points to d subspaces is done dynamically and hence the index is more adaptive and dynamic, (2) at most d subqueries are required, (3) the computation of iMinMax value is simpler.

The VA-file (vector approximation file) [15] is based on the idea of object approximation by mapping a coordinate to some value that reduces the storage requirement. The basic idea is to divide the data space into 2^b hyperrectangular cells, where b is the tunable number of bits used for representation. For each dimension i, b_i bits are used, and 2^{b_i} slices are generated in such a way that all slices are equally full. The data space consists of 2^b hyperrectangular cells, each of which can be represented by a unique bit string of length b. A data point is then approximated by the bit string of the cell it falls into. To perform a point or range query, the entire approximation file must be sequentially scanned. Objects whose bit string satisfies the query must be retrieved and checked. Typically, the VA-file is much smaller than the vector file and hence is far more efficient than a direct sequential scan of the data file and the variants of the R-tree. However, the performance of the VA-file is likely to be affected by data distributions and hence the false drop rate, the number of dimensions, and the volume of data.

The iDistance [17] was specifically designed for K-nearest neighbor (KNN) search in a high-dimensional space. The basic idea of iDistance is to partition the data points and select a reference point for each partition. The data points in each cluster are transformed into a single-dimensional space based on their similarity with respect to the reference point. This allows the data points to be indexed using a B$^+$-tree structure and KNN search to be performed using one-dimensional range search. The KNN search starts with a small query sphere. For each data space defined by the reference point, if it intersects the query sphere, a range query is performed. This process is repeated with increasingly larger query spheres until KNNs are retrieved. The choice of partition and reference point provides the iDistance technique with degrees of freedom most other KNN techniques do not have. Performance studies have shown that iDistance outperforms the recent A-tree [14] and sequential scan significantly. iDistance and iMinMax(θ) are similar only in their transformation and their use of B$^+$-tree as the base index. While iMinMax(θ) is attribute based, iDistance is metric based, and therefore iDistance cannot be used to support attribute-based retrieval.

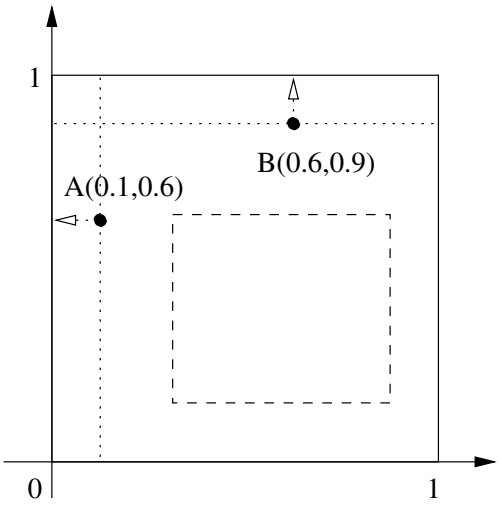


Fig. 2. Example of the edge concept in two-dimensional space

3 Indexing on the edges

In a multidimensional range search, all values of all dimensions must satisfy the query range along each dimension. If any of them fails, the data point will not be in the answer set. Based on this observation, a straightforward approach is to index on a small, but fixed, subset of the dimensions. However, the effectiveness of such an approach depends on the data distribution of the selected dimensions. Our preliminary study on indexing one single dimension showed that the approach can perform worse than sequential scanning. This led us to examine novel techniques that index on the "edges". An "edge" of a data point refers to the maximum or minimum value among all the dimensions of the point. The "edge" of a data point is also the attribute that is closer to the data space edge than other attributes. We shall refer to the edge with the maximum value as the *Max edge* and the edge with the minimum value as the *Min edge*. Figure 2 illustrates the edge concept in two-dimensional space. Here, for point A (0.1, 0.6), its edge is 0.1 as 0.1 is closer to the x-axis than 0.6 is to the y-axis. On the other hand, the edge of point B (0.6, 0.9) is 0.9. We note that point A is mapped to the Min edge and point B to the Max edge. We also note that a data point whose "edge" is not included in a query range cannot be an answer of the query.

The basic idea of iMinMax(θ) is to use either the values of the *Max edge* or the values of the *Min edge* as the representative index keys for the points. Because the transformed values can be ordered and range queries performed on the transformed (single-dimensional) space, we can employ single-dimensional indexes to index the transformed values. In our research, we employed the B$^+$-tree structure since it is supported by all commercial DBMSs. Thus, the iMinMax method can be readily adopted for use.

In the following discussion, we consider a unit d-dimensional space, i.e., points are in the space ([0,1],[0,1],...,[0,1]). We denote an arbitrary data point in the space as $x = (x_1, x_2, \ldots, x_d)$. Let $x_{max} = \max_{i=1}^{d} x_i$

and $x_{min} = \min_{i=1}^{d} x_i$ be, respectively, the maximum value and minimum value among the dimensions of the point. Moreover, let d_{max} and d_{min} denote the dimensions at which the maximum and minimum values occur. Let the range query be q = ($[x_{11}, x_{12}]$, $[x_{21}, x_{22}]$, ..., $[x_{d1}, x_{d2}]$). Let $ans(q)$ denote the answers produced by evaluating a query q.

3.1 Mapping high-dimensional data to single-dimensional space

iMinMax(θ) adopts a simple mapping function that is computationally inexpensive. The data point x is mapped to a point y over a single-dimensional space as follows:

$$y = \begin{cases} d_{min} \times c + x_{min} & \text{if } x_{min} + \theta < 1 - x_{max} \\ d_{max} \times c + x_{max} & \text{otherwise} \end{cases}$$

where θ is a real number and c a positive constant.

First, we note that θ plays an important role in influencing the number of points falling on each index hyperplane. In fact, it is the tuning knob that affects the hyperplane an index point should reside on. Take a data point (0.2, 0.75) in two-dimensional space for example; with $\theta = 0.0$, the index point will reside on the Min edge. Setting θ to 0.1 will push the index point to reside on the Max edge. The higher the value of $\theta \geq 0$, the more the transformation function is biased toward the Max edge. When $\theta = 0.1$, the Max edge has a preference of about 10% more. Similarly, we can "favor" the transformation to the Min edge with $\theta < 0$. In fact, at one extreme, when $\theta \geq 1.0$, the transformation maps all points to their Max edge; and by setting $\theta \leq -1.0$, we always pick the value at the Min edge as the index key. For simplicity, we shall denote the former extreme as iMax, the latter extreme as iMin, and any other variation as iMinMax (dropping θ unless its value is critical).

Second, we note that the transformation function actually comprises two components: the *dimension* and the *value* of the dimension. The first component is used to split the (single-dimensional) data space into different partitions based on the dimension with the largest value or smallest value. This is done to scatter points with the same edge values (but from different edges) to minimize false drops. In our case, the Max or Min edge serves the purpose. The second component provides an ordering within each partition. Thus, the overall effect on the data points is that all points whose edge is in dimension i will be mapped into the range $[i \times c, i \times c + 1]$. Note that c is used to keep the partitions apart to minimize their overlaps. To avoid any overlap, c should be larger than 1. If c is less than or equal to 1, the correctness of iMinMax is not affected; however, iMinMax's efficiency may suffer as a result of excessive false drops. For ease of presentation, in the rest of this paper we shall assume $c = 1$ and exclude c from the formulas.

Finally, the unique tunable feature facilitates the adaptation of iMinMax(θ) to data sets of different distributions (uniform or skewed). In cases where data points are skewed toward certain edges, we may "scatter" these points to other edges to evenly distribute them by making a choice between d_{min} and d_{max}. Statistical information such as the number of index points can be used for such a purpose. Alternatively, one can use either the information regarding data distribution or information collected to categorically adjust the partitioning.

3.2 Mapping range queries

Range queries on the original d-dimensional space have to be transformed to the single-dimensional space for evaluation. In iMinMax(θ), the original query on the d-dimensional space is mapped into d subqueries – one for each dimension. Let us denote the subqueries as q_1, q_2, ..., q_d, where $q_i = [x_{i1}, y_{i2}]$ $1 \leq i \leq d$. For the j-th query subrange in q, $[x_{j1}, x_{j2}]$, we have

$$q_j = \begin{cases} [j + \max_{i=1}^{d} x_{i1},\ j + x_{j2}] & \text{if } \min_{i=1}^{d} x_{i1} \\ & \quad +\theta \geq 1 - \max_{i=1}^{d} x_{i1} \\ [j + x_{j1},\ j + \min_{i=1}^{d} x_{i2}] & \text{if } \min_{i=1}^{d} x_{i2} \\ & \quad +\theta < 1 - \max_{i=1}^{d} x_{i2} \\ [j + x_{j1},\ j + x_{j2}] & \text{otherwise.} \end{cases}$$

The union of the answers from all subqueries provides the candidate answer set from which the query answers can be obtained, i.e., $ans(q) \subseteq \cup_{i=1}^{d} ans(q_i)$. We shall now prove some interesting results.

Theorem 1. *Under the iMinMax(θ) scheme, $ans(q) \subseteq \cup_{i=1}^{d} ans(q_i)$. Moreover, there does not exist $q_i' = [x_{i1}', x_{i2}']$, where $x_{i1}' > x_{i1}$ or $x_{i2}' < x_{i2}$, for which $ans(q) \subseteq \cup_{i=1}^{d} ans(q_i')$ always holds. In other words, q_i is "optimal" and narrowing its range may miss some of q's answers.*

Proof. For the first part, we need to show that any point x that satisfies q will be retrieved by some $q_i, 1 \leq i \leq d$. For the second part, we only need to show that some points that satisfy q may be missed. The proof comprises three parts corresponding to the three cases in the range query mapping function.

Case 1: $\min_{i=1}^{d} x_{i1} + \theta \geq 1 - \max_{i=1}^{d} x_{i1}$

In this case, all the answer points that satisfy the query q have been mapped to the Max edge, i.e., a point x that satisfies q is mapped to x_{max}, and would have been mapped to the d_{max}-th dimension, and has index key of $d_{max} + x_{max}$. The subquery range for the d_{max}-th dimension is $[d_{max} + \max_{i=1}^{d} x_{i1}, d_{max} + x_{d_{max}2}]$. Since x satisfies q, we have $x_i \in [x_{i1}, x_{i2}], \forall i, 1 \leq i \leq d$. Moreover, we have $x_{max} \geq x_{i1}$ $\forall i, 1 \leq i \leq d$. This implies that $x_{max} \geq \max_{i=1}^{d} x_{i1}$ $\forall i, 1 \leq i \leq d$. We also have $x_{max} \leq x_{d_{max}2}$. Therefore, we have $x_{max} \in [\max_{i=1}^{d} x_{i1},\ x_{d_{max}2}]$, i.e., x can be retrieved using the d_{max}-th subquery. Thus, $ans(q) \subseteq \cup_{i=1}^{d} ans(q_i)$.

Now, let $q_i' = [l_i' + \epsilon_l, h_i' - \epsilon_h]$ for some $\epsilon_l > 0$ and $\epsilon_h > 0$. Consider a point $z = (z_1,\ z_2,\ \ldots, z_d)$ that satisfies q. We note that if $l_i < z_{max} < l_i + \epsilon_l$, then we will miss z if q_i' has been used. Similarly, if $h_i - \epsilon_h < z_{max} < \max_{i=1}^{d} x_{i2}$, then we will also miss z if q_i' has been used. Therefore, no q_i' provides the tightest bound that guarantees that no points will be missed.

Case 2: $\min_{i=1}^{d} x_{i2} + \theta < 1 - \max_{i=1}^{d} x_{i2}$

This case is the inverse of case 1, i.e., all points in the query range belong to the Min edge. As such, we can apply similar logic. We shall not show the proof here. We refer the interested reader to [16].

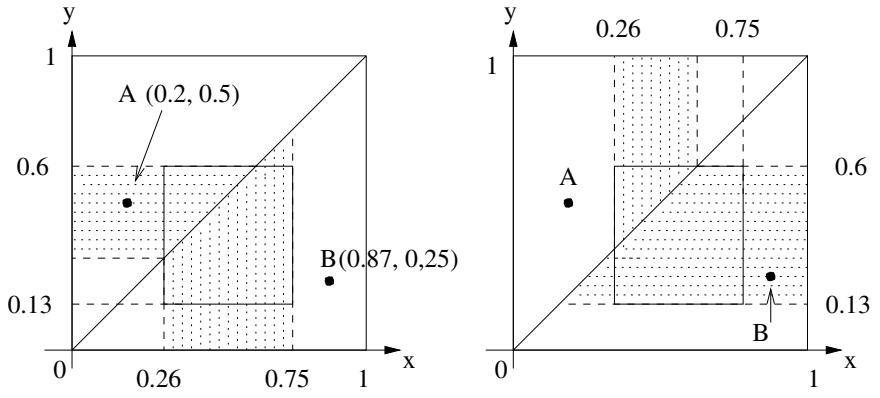


Fig. 3. Sample search space for two-dimensional space

Case 3

In case 3, the answers of q may be found in both the Min edge and the Max edge. Given a point x that satisfies q, we have $x_i \in [x_{i1}, x_{i2}], \forall i, 1 \leq i \leq d$. We have two cases to consider. In the first case, x is mapped to the Min edge, its index key is $d_{min} + x_{min}$, and it is indexed on the d_{min}-th dimension. To retrieve x, we need to examine the d_{min}-th subquery, $[d_{min} + x_{d_{min}1},\ d_{min} + x_{d_{min}2}]$. Now, we have $x_{min} \in [x_{d_{min}1}, x_{d_{min}2}]$ (since x is in the answer) and hence the d_{min}-th subquery will be able to retrieve x. The second case, which involves mapping x onto the Max edge, can be similarly derived. Thus, $ans(q) \subseteq \cup_{i=1}^{d} ans(q_i)$.

Now, let $q_i' = [l_i' + \epsilon_l, h_i' - \epsilon_h]$ for some $\epsilon_l > 0$ and $\epsilon_h > 0$. Consider a point $z = (z_1,\ z_2,\ \ldots, z_d)$ that satisfies q. We note that if $l_i < z_{max} < l_i + \epsilon_l$, then we will miss z if q_i' has been used. Similarly, if $h_i - \epsilon_h < z_{max} < h_i'$, then we will also miss z if q_i' has been used. Therefore, no q_i' provides the tightest bound that guarantees that no points will be missed. □.

We would like to point out that in an actual implementation, the leaf nodes of the B$^+$-tree will contain the high-dimensional point, i.e., even though the index key on the B$^+$-tree is only single-dimensional, the leaf node entries contain the triple (x_{key}, x, ptr), where x_{key} is the single-dimensional index key of point x and ptr is the pointer to the data page containing other information that may be related to the high-dimensional point. Therefore, the false drop of Theorem 1 affects only the vectors used as index keys rather than the actual data themselves.

Theorem 2. *Given a query q and the subqueries $q_1,\ q_2,\ \ldots,\ q_d$, q_i need not be evaluated if any of the followings holds:*

(i) $\min_{j=1}^{d} x_{j1} + \theta \geq 1 - \max_{j=1}^{d} x_{j1}$ *and* $h_i < \max_{j=1}^{d} x_{j1}$.

(ii) $\min_{j=1}^{d} x_{j2} + \theta < 1 - \max_{j=1}^{d} x_{j2}$ *and* $l_i > \min_{j=1}^{d} x_{j2}$.

Proof. Consider the first case: $\min_{j=1}^{d} x_{j1} + \theta \geq 1 - \max_{j=1}^{d} x_{j1}$ and $h_i < \max_{j=1}^{d} x_{j1}$. The first expression implies that all the answers for q can only be found in the Max edge. We note that the point with the smallest maximum value that satisfies q is $\max_{j=1}^{d} x_{j1}$. This implies that if $h_i < \max_{j=1}^{d} x_{j1}$, then the answer set for q_i will be an empty set. Thus, q_i need not be evaluated.

The second expression means that all the answers for q are located in the Min edge. The point with the largest minimum value that satisfies q is $\min_{j=1}^{d} x_{j2}$. This implies that if $l_i > \min_{j=1}^{d} x_{j2}$, then the answer set for q_i will be empty. Thus, q_i need not be evaluated. □

Example 1. Let $\theta = 0.5$. Consider the range query ([0.2,0.3], [0.4,0.6]) in two-dimensional space. Since $0.2 + 0.5 > 1 - 0.4 = 0.6$, we know that all points that satisfy the query fall on the Max edge. This means that the lower bound for the subqueries should be 0.4, i.e., the two subqueries are, respectively, [0.4,0.3] and [0.4,0.6]. Clearly, the first subquery range is not valid as the derived lower bound is larger than the upper bound. Thus, it need not be evaluated because no points will satisfy the query.

As a consequence of the mapping strategy, where a d-dimensional space is partitioned into k-dimensional subspaces, we need to search at most d subspaces, and hence the number of subqueries is bounded by d, as formally stated below.

Theorem 3. *Given a query q and the subqueries $q_1,\ q_2,\ \ldots,\ q_d$, at most d subqueries need to be evaluated.*

Proof. The proof is straightforward and follows from Theorem 2. □

From Theorems 2 and 3 we have a glimpse of the effectiveness of iMinMax(θ). In fact, for very high-dimensional spaces, we can expect significant savings from the pruning of subqueries.

Example 2. In this example, we illustrate how iMinMax can keep out points from the search space. Figure 3 shows an example. Here we have two points A(0.2,0.5) and B(0.87,0.25) in two-dimensional space. If we employ either iMax or iMin, at least one false drop will occur. On the other hand, using iMinMax(0.5) effectively keeps both points out of the search space.

4 Operations on iMinMax(θ)

In our implementation of iMinMax(θ), we have adopted the B$^+$-tree [13] as the underlying single-dimensional index struc-

Algorithm PointSearch

Input: point p, θ, root of the B$^+$-tree R
Output: tuples matching p

```
1.    x_p ← transform(p, θ)
2.    l ← traverse(x_p, R)
3.    if x_p is not found in l
4.        return (NULL)
5.    else
6.        S ← ∅
7.        for every entry (x_v, v, ptr) in l with key x_v == x_p
8.            if v == p
9.                tuple ← access(ptr)
10.               S ← S∪ tuple
11.       if l's last entry contains key x_p
12.           l ← l's right sibling
13.           goto 7
14.       return (S)
```

Fig. 4. Point search algorithm

Algorithm RangeSearch

Input: range query $q = ([x_{11}, x_{12}], [x_{21}, x_{22}], \ldots)$, root of the B$^+$-tree R
Output: answer tuples to the range query

```
1.  S ← ∅
2.  for (i = 1 to d)
3.      q_i ← transform(r, i)
4.      if NOT(pruneSubquery(q_i, q))
5.          l ← traverse(x_i1, R)
6.          for every entry (x_v, v, ptr) in l with key x_v ∈ [x_i1, x_i2]
7.              if v ∈ q
8.                  tuple ← access(ptr)
9.                  S ← S∪ tuple
10.         if l's last entry contains key x < x_i2
11.             l ← l's right sibling
12.             goto 6
13. return (S)
```

Fig. 5. Range search algorithm

ture. However, for greater efficiency, leaf nodes also store the high-dimensional vector, i.e., leaf node entries are of the form (key, v, ptr), where key is the single-dimensional key, v is the high-dimensional vector whose transformed value is key, and ptr is the pointer to the data page containing information related to v. Keeping v at the leaf nodes can minimize page accesses to nonmatching points. We note that multiple high-dimensional keys may be mapped to a single key value.

The search, insert, and delete algorithms are similar to the B$^+$-tree algorithms. The additional complexity arises as we have to deal with multiple subqueries in the single-dimensional space and the additional high-dimensional key (besides the single-dimensional key value). As such, we shall just present the search algorithms and omit the algorithmic descriptions for insert and delete operations.

4.1 Point search algorithm

In point search, a point p is issued and all matching tuples are to be retrieved. Clearly, by the transformation only one partition needs to be searched – the partition that corresponds to either the maximum attribute value (Max edge) or minimum value (Min edge), depending on the value of θ. However, if θ was tuned during the life span of the index for performance purposes, both the maximum and minimum attribute values of p have to be used for searching.

The algorithm is summarized in Fig. 4. Based on θ, the search algorithm first maps p to the single-dimensional key, x_p, using the function **transform(point, θ)** (line 1). For each query, the B$^+$-tree is traversed (line 2) to the leaf node where x_p may be stored. If the point does not exist, then a NULL value is returned (lines 3-4). Otherwise, for every matching x_p value, the high-dimensional key of the data entry is compared with p for a match. Those that match are accessed using the pointer value (lines 7–13); otherwise, they are ignored. We note that it is possible for a sequence of leaf nodes to contain matching key values, and hence they all have to be examined. The final answers are then returned (line 14).

4.2 Range search algorithm

Range queries are slightly more complicated than point search. Figure 5 shows the algorithm. Unlike point queries, a d-dimensional range query r is transformed into d subqueries (lines 2,3). The i-th subquery is denoted as $q_i = [x_{i1}, x_{i2}]$. Next, routine **pruneSubquery** is invoked to check if q_i can be pruned (line 4). This is based on Theorem 2. If it can be, then it is ignored. Otherwise, the subquery is evaluated as follows (lines 5–12). The B$^+$-tree is traversed using the lower bound of the subquery to the appropriate leaf node. If there are no points in the range of q_i, then the subquery stops. Otherwise, for every $x \in [x_{i1}, x_{i2}]$, the high-dimensional key of the data is checked against q to see if it is contained by q. Those that fall within the range are accessed using the pointer value. As in point search, multiple leaf pages may have to be examined. Once all subqueries have been evaluated, the final answers are then returned (line 13).

5 A performance study

In this section, we shall present an extensive performance study to evaluate the effectiveness of iMinMax(θ). The objectives of the study are threefold:

1. To verify the correctness of the iMinMax(θ) and to study the effect of θ.
2. To study the effect of external factors such as buffering and different data/query distributions on the index.
3. To make an empirical comparison with existing methods; in particular, we have compared iMinMax(θ) with the VA-file (vector-approximation scheme) [15] and the Pyramid scheme [2], as well as the simple sequential scan strategy.

5.1 Experiment setup

We implemented iMinMax(θ), the VA-file, and the Pyramid technique in C on a SUN Sparc Workstation and used the

Table 1. Parameters and their values

Parameter	Default values	Variations
System parameters		
Page size	4K page	
Index node size	4K page	
Buffer size	128 pages	64, 256, 512
Database parameters		
No. of tuples	500,000	100,000–500,000
No. of dimensions	32	8–128
Domain of dimensions	[0..1]	
Data distribution	Uniform	Exponential, normal
Query parameters		
Range query selectivity	0.1%	1%,5%,10%
No. of queries	500	
Query distribution	Uniform	Exponential, normal

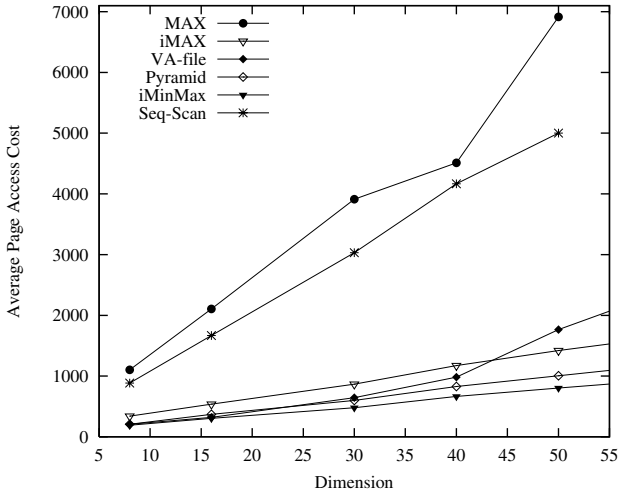


Fig. 6. Effect of dimensionality on uniformly distributed data set

B^+-tree as the single-dimensional base index structure for the Pyramid and iMinMax(θ) methods. For the VA-file, we used bit strings of length $b = 4$, as recommended by the inventors.

Each index page is 4 KB. We also implemented a known priority-based buffer replacement strategy for hierarchical indexes [7] to manage buffer space. The replacement strategy assigns priority to each node as it traverses, and as it backtracks in the tree traversal the priority is reassigned as the node is no longer useful or its likelihood of being rereferenced is not high. Such a strategy has been shown to work well for hierarchical tree traversal.

To study the efficiency of iMinMax(θ), experiments on various data sets, especially nonuniformly distributed and clustered data sets, are necessary. In our experiments, apart from the uniform data set, we also generate skewed data sets that follow normal and exponential distributions. Table 1 summarizes the parameters and default values used in the experiments.

Similarly, to test the performance of the indexes against queries, various types of range queries are used. We, however, assume that the range queries follow the same distribution as that of the targeted data set. The default range queries conducted on uniformly distributed data sets are queries with fixed selectivity of 0.1% whose centers are randomly picked from within the data space. The default range queries conducted on normally distributed data sets are queries with range side = 0.4 and whose centers are picked following a normal distribution. The default range queries conducted on exponentially distributed data sets are queries with range side = 0.4 whose centers are selected based on an exponential distribution.

We conducted extensive experiments. In each experiment, we ran 500 range queries. Each query is a hypercube and has a default selectivity of 0.1% of the domain space ([0,1],[0,1],. . .,[0,1]). The query width is the d-th root of the selectivity: $\sqrt[d]{0.001}$. As an indication of how large the width of a fairly low selectivity can be, the query width for 40-dimensional space is 0.841, which is much larger than half of the extension of the data space along each dimension. Different query sizes will be used for nonuniform distributions. The default number of dimensions used is 30. Each I/O corresponds to the retrieval of a 4-KB page. The average I/O cost of the queries is used as the performance metrics. For the experiments where we did not want to have the buffering effect on the page I/O, we turned off the buffer. In such cases, every page touched incurs an I/O.

5.2 Effect of the number of dimensions

In the first set of experiments, we vary the number of dimensions from 8 to 50. The data set is uniformly distributed over the domain space. There are a total of 100K points.

In the first experiment, besides the Pyramid technique and VA-file, we also compare iMinMax(θ) with the Max scheme and the sequential scan (seq-scan) technique. The Max scheme is a simple scheme that maps each point to its maximum value. However, the transformed space is not partitioned. Moreover, two variations of iMinMax(θ) are used, namely, iMax (i.e., $\theta = 1$) and iMinMax($\theta = 0.0$) (denoted as iMinMax). Figure 6 shows the results. First, we note that both the Max and seq-scan techniques perform poorly, and their I/O costs increase with increasing dimensionality. Max performs slightly worse because of the additional internal nodes to be accessed and the high number of false drops.

Second, while the number of I/Os for iMinMax, iMax, Pyramid, and VA-file also increases with an increasing number of dimensions, it grows at a much slower rate. Third, we see that iMinMax performs the best, with Pyramid and VA-file following closely, and iMax performing mostly worse than Pyramid and VA-file. iMinMax outperforms iMax, Pyramid, and VA-file since its search space touches fewer points and some of the range subqueries can be pruned.

In a typical application, apart from the index attributes, there are many more large attributes that make sequential scan of an entire file non-cost-effective. Instead, a feature file that consists of vectors of index attribute values is used to filter out objects (records) that do not match the search condition. However, we note that for queries that entail retrieval of a large proportion of objects, direct sequential scan may still be cost effective. The data file of the iMinMax technique can be clustered based on the leaf nodes of its B^+-tree to reduce random reads. The clusters can be formed in such a way that they allow easy insertion of objects and expansion of their extent. Other optimizations such as those at the physical level can make the B^+-tree behave like an index sequential file.

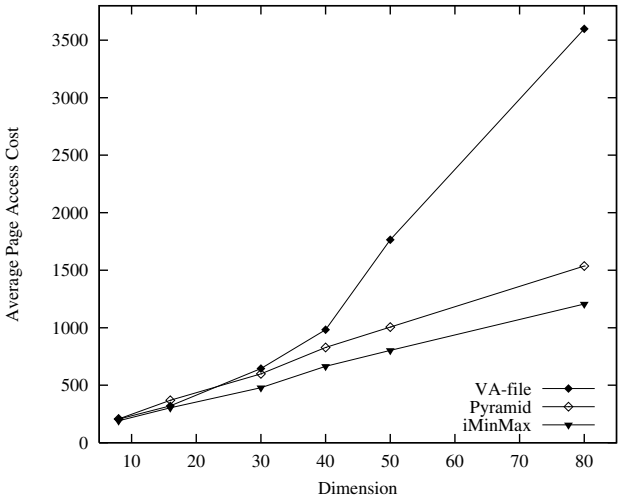


Fig. 7. Comparing VA-file, iMinMax, and Pyramid schemes

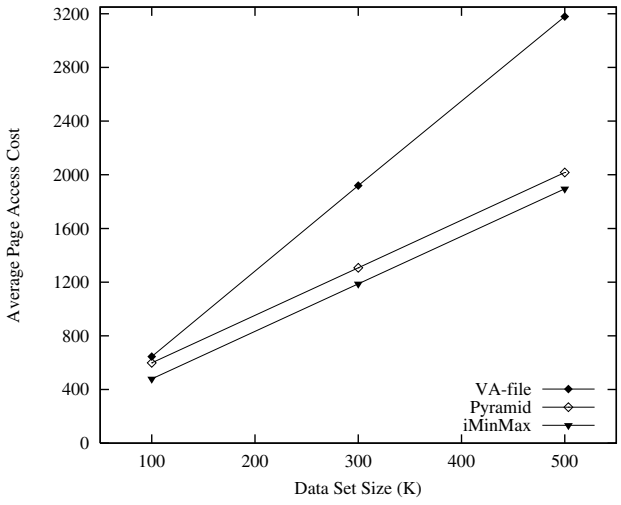


Fig. 8. Effect of varying data set sizes

Based on the above arguments and the experimental results, we expect iMinMax, the Pyramid technique, and VA-file to be competitive. Thus, for subsequent experiments, we shall restrict our study to these techniques only.

We further evaluated Pyramid, VA-file, and iMinMax, and the results are shown in Fig. 7. We observe that iMinMax remains superior and can outperform Pyramid by up to 25% and VA-file by up to 250%.

5.3 Effect of data size and query selectivity

In this set of experiments, we study the effect of data set size and the query selectivity. For both studies, we fix the number of dimensions at 30. Figure 8 shows the results when we vary the data set sizes from 100K to 500K points. Figure 9 shows the results when we vary the query selectivities from 0.01% to 10%.

As expected, iMinMax, Pyramid, and VA-file all incurred higher I/O costs with increasing data set sizes as well as query selectivities. As before, iMinMax remains superior to the Pyramid scheme and VA-file scheme. It is interesting to note that the relative difference between iMinMax and the Pyramid scheme seems to be unaffected by data set size and query selectivity. Due to the uniform distributions, data points are evenly distributed among partitions for both methods. Although the iMinMax has only d partitions, the points are distributed along the Max and Min edges. The improvement of iMinMax stems from its reduced number of subqueries compared to the Pyramid scheme. The performance of VA-file is worse than both Pyramid and iMinMax. In subsequent comparisons on skewed data sets, we shall focus on iMinMax and Pyramid schemes.

5.4 Effect of skewed data distributions

In this study we investigate the relative performance of iMinMax and Pyramid on skewed data distributions. Here, we show the results on two distributions, namely, skewed normal and skewed exponential. Figure 10 illustrates these two skewed data distributions in two-dimensional space.

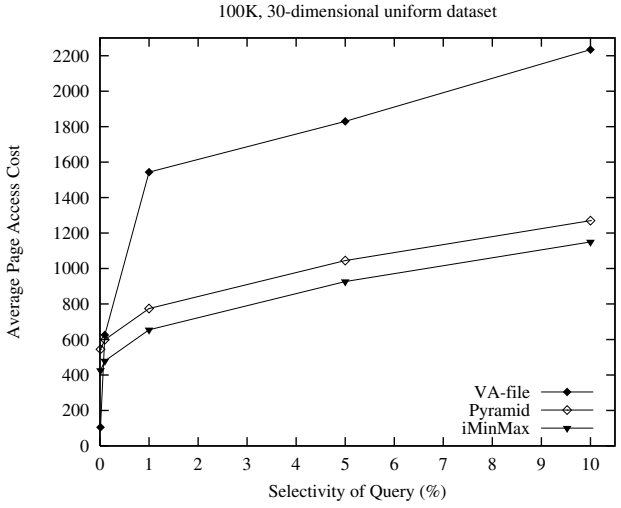


Fig. 9. Effect of varying query selectivity (100K data set)

The first set of experiments studies the effect of θ on skewed normal distribution. For normal distribution, the closer the data center is to the cluster center, the more we can keep points evenly assigned to each edge. For queries that follow the same distribution, the data points will have the same probability of being kept far from the query cube. In these experiments, we fix the query width of each dimension at 0.4.

Figure 11a shows the results for 100K 30-dimensional points. First, we observe that for iMinMax there exists a certain optimal θ value that leads to the best performance. Essentially, θ "looks out for" the center of the cluster. Second, iMinMax can outperform the Pyramid technique by a wide margin (more than 50%!). Third, we note that iMinMax can perform worse than the Pyramid scheme. This occurs when the distributions of points to the edges become skewed and a larger number of points have to be searched. Because of the above points, we note that it is important to fine-tune θ for different data distributions in order to obtain optimal performance. The nice property is that this tuning can be easily performed by varying θ.

In Fig. 11b, we have the results for 500K 30-dimensional points. As in the earlier experiment, iMinMax's effectiveness

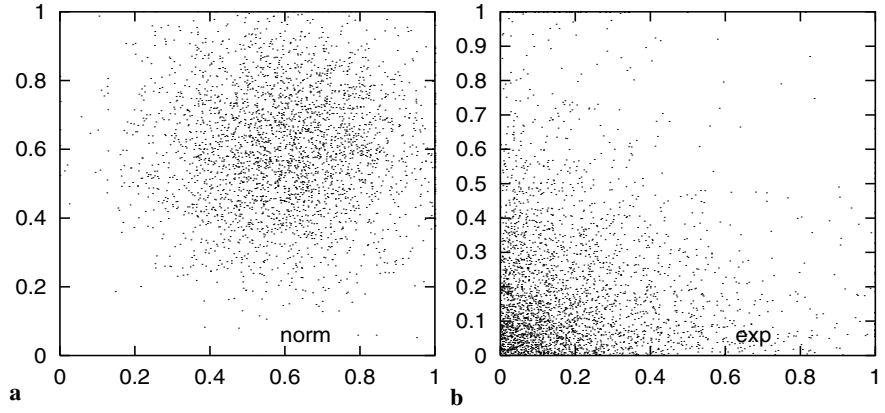
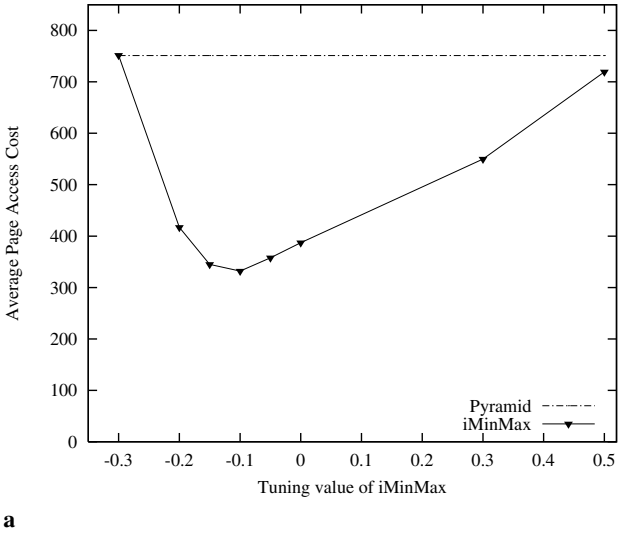


Fig. 10. Skewed data distributions. **a** Normal distribution. **b** Exponential distribution



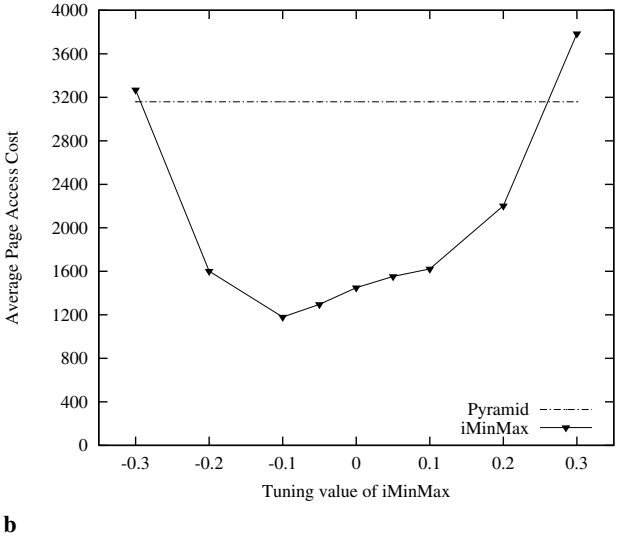


Fig. 11a,b. Skewed normal data set. **a** 100K points. **b** 500K points

depends on the θ value set. We observe that iMinMax performs better than Pyramid over a wider range of tuning factors and over a wider margin (more than 66%).

The second set of experiments looks at the relative performance of the schemes for skewed exponential data sets.

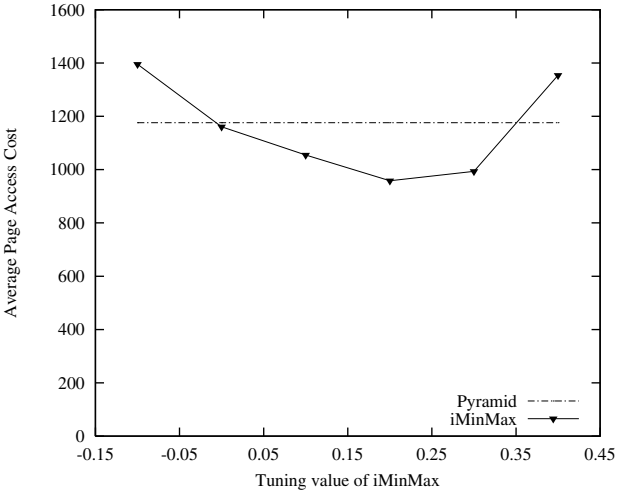
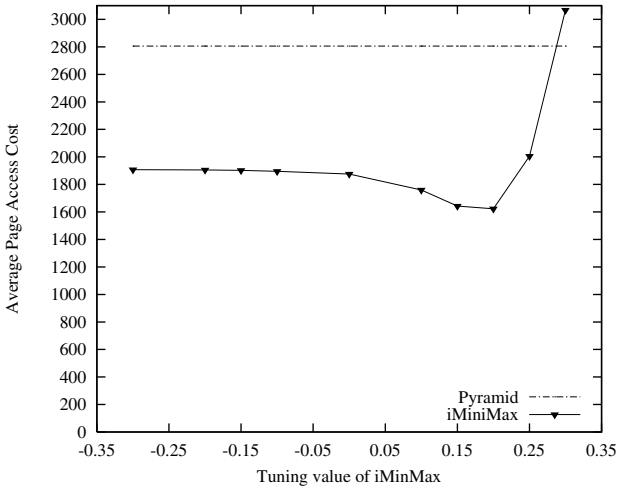


Fig. 12. Skewed exponential data sets (500K points)

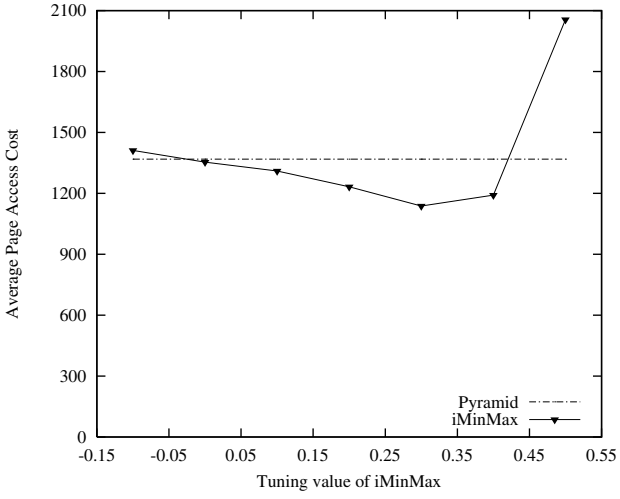
As above, we fix the query width of each dimension at 0.4. For exponential distribution (we choose to be exponential to small values), many dimensions will have small values, and a small number of them will have large values. Thus, many data points will have at least one large value. Because many of the dimensions have small values, the data points tend to lie close along the edges of the data space. We note that exponential data distributions can differ greatly from one another. They are more likely to be close along the edges or close to the different corners depending on the number of dimensions that are skewed to be large or small. For a range query that is generated using an exponential distribution, its subqueries will mostly be close to the low corner. Therefore, tuning the index keys toward large values is likely to filter out more points from the query.

Figure 12 shows the results for 500K 30-dimensional points on a skewed exponential distribution. The results are similar to that of the normal distribution experiments – iMinMax is optimal at certain θ values. For the results on 100K 30-dimensional data points, although the iMinMax shows similar behavior, the gain is not as impressive due to the skewness of distribution and smaller data set, and hence we omit the graph here.

We also tested iMinMax with two very skewed data sets. One of them has very skewed data pointers at one corner of



a



b

Fig. 13a,b. Highly skewed data sets. **a** Skewed at one corner of data set. **b** Skewed at some corners of data set

the data space. The other also has very skewed data pointers, but clustered at different corners of the data set. These data sets contain 500K 30-dimensional data points. And the range queries used are 0.4 wide. Figure 13 shows that tuning iMinMax can improve the performance well on very skewed data sets.

5.5 Effect of buffer space

In this section, we shall see the effect of buffer space on the performance of iMinMax. We use the buffer sizes of 0, 64, 128, 256, and 512 pages. Each page is 4 KB. Under the buffering strategy used, frequently used pages such as those near the root have higher priority and hence are kept longer in the buffer. We use uniform data sets with 100K data points and set the number of dimensions, respectively, at 8, 16, 30, 40, 50, and 80. Figure 14 shows the effect of buffer space.

The iMinMax method is built on top of an existing B^+-tree, and the traversal is similar to that of the conventional B^+-tree, although each query translates to d subinterval searches. It should be noted that the traversal paths of the d subqueries

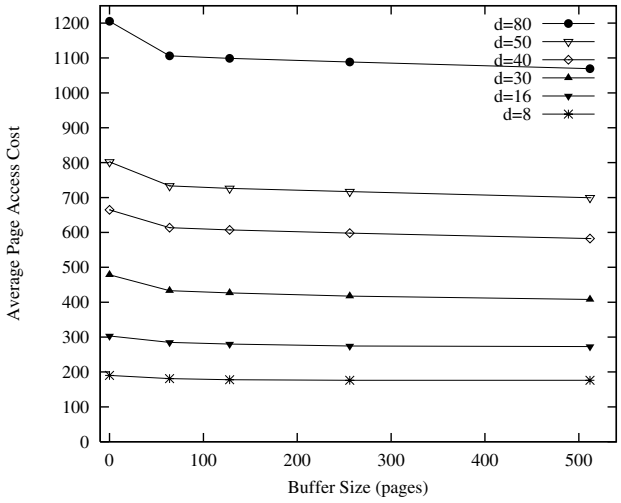


Fig. 14. Effect of buffer space

generated by iMinMax(θ) do not overlap and hence share very few common internal nodes. This is also true of the subqueries generated by the Pyramid technique. Nevertheless, as in any database applications, buffering reduces the number of pages that need to be fetched due to rereferencing. The performance results show such gain, albeit marginal, and it decreases as the buffer size increases toward a saturated state. The marginal gain is due to little overlap between the d range queries. Each of the d subqueries goes down from the root node along a single path to a leaf node and scans the leaf nodes rightward till a value outside the search range is encountered. For each query, the root node has to be fetched at most once, and some nodes close to the root may be rereferenced without incurring additional page accesses. Leaf nodes are not rereferenced in any of the d subqueries, and the priority-based replacement strategy replaces them as soon as they are unpinned. Therefore, no significant savings can be obtained by buffering leaf pages in the iMinMax method. This result actually implies that iMinMax can perform well even given a small main memory.

5.6 CPU cost

While the ratio between the access time of memory and hard disk remains at about 5 orders of magnitude, the CPU cost is an important criterion to consider in designing and selecting an index. Indeed, for some indexes, the cost has shifted from I/O to CPU cost.

Figure 15 shows the CPU costs of iMinMax against that of the Pyramid method, VA-file, and linear scan. While linear scan (seq-scan) incurs less seek time and disk latency, the number of pages scanned remains large and the entire file has to be checked line by line. This explains why linear scan does not perform well as a whole. iMinMax performs better than the Pyramid method, and the gain widens as the number of dimensions increases. When the number of dimensions is small, the VA-file is more efficient than the Pyramid method. However, as the number of dimensions increases, its CPU costs increase at a much faster rate than those of the Pyramid method. Again, this is attributed to more comparisons and also random accesses when fetching data objects after checking.

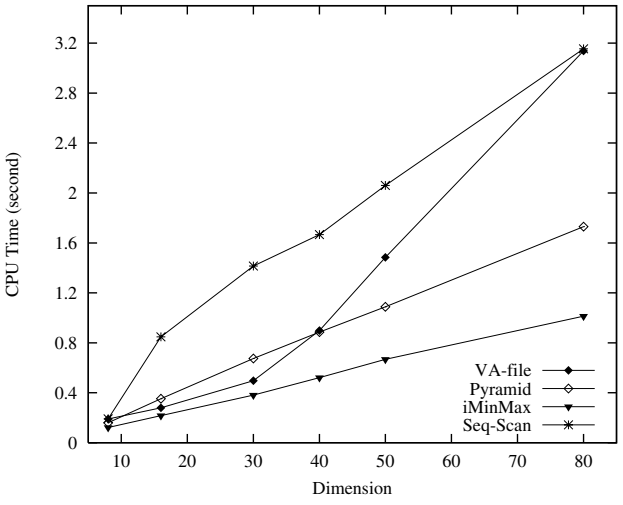


Fig. 15. Comparing CPU cost of iMinMax, Pyramid, and VA-file

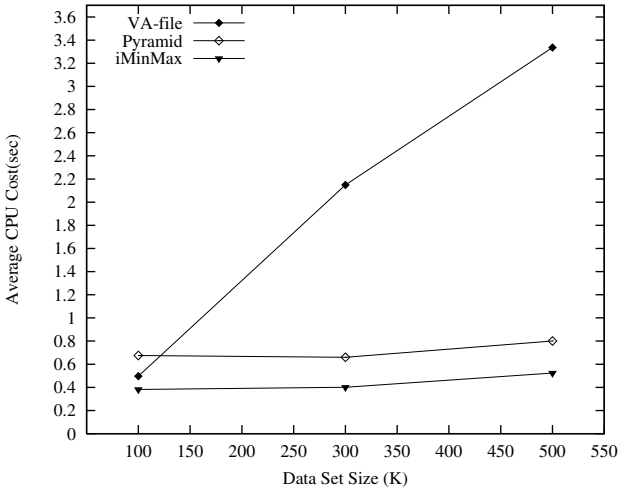


Fig. 16. Effect of varying data set sizes on CPU cost

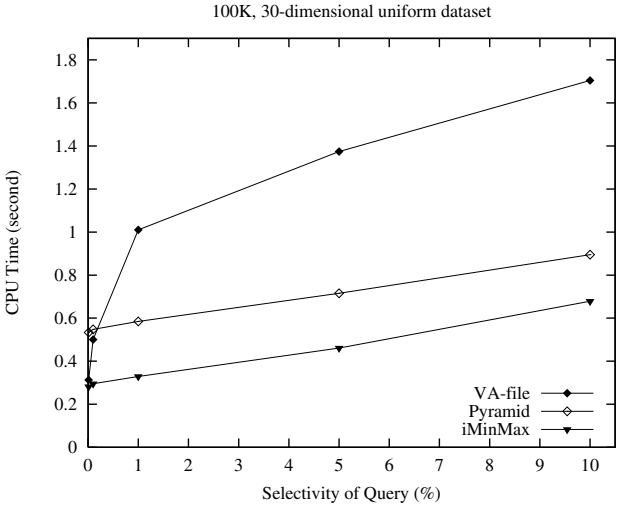


Fig. 17. Effect of query selectivity on CPU cost (100K data set)

Figure 16 shows the effect of data volume on CPU costs. As the number of data points increases, the performance of the VA-file degrades rapidly. The reasons are similar to those of the previous experiment. iMinMax shows consistent performance and is more efficient than both Pyramid method and VA-file and is not sensitive to data volume. The increase in data size does not increase the height of the B^+-tree substantially, and the approach can effectively filter away objects that are not in the answer.

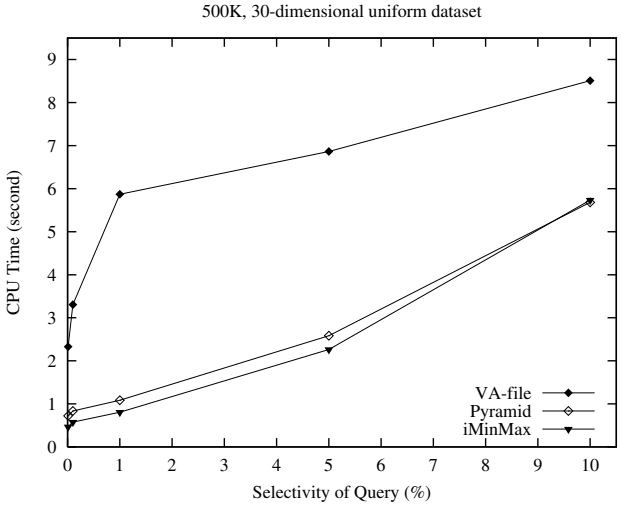


Fig. 18. Effect of query selectivity on CPU cost (500K data set)

Figure 17 summarizes the effect of query selectivity on CPU costs. All methods are affected by an increase in query size, as this involves more fetching of objects and checking. The result shows the superiority of iMinMax over the other two methods.

We did the same testing on a 500K data set, and the result exhibits a trend similar to that of the 100K data set. Figure 18 summarizes the results. However, when the data size is large, the gain of iMinMax over the Pyramid method decreases with increasing selectivity. With large selectivity, the number of objects that need to be examined increases and hence the CPU cost incurred also increases.

5.7 Summary

The above experiments make it clear that the iMinMax(θ) is an efficient indexing method for supporting range query. It outperforms the Pyramid method and VA-file by a wide margin. The performance gain is remarkably significant for skewed data sets and large data volume.

The advantages of the iMinMax(θ) over the Pyramid method are threefold: the iMinMax(θ) is simpler in design principle and hence less complex computationally; the iMinMax(θ) requires d queries compared to $2d$ queries of the Pyramid method; the iMinMax(θ) is more dynamic and can be tuned without reconstruction of the index and hence is more adaptable for skewed data sets.

In summary, iMinMax(θ) is a flexible, adaptive, and efficient indexing method. More importantly, it can be integrated into existing DBMSs at the application layer by coding the mapping functions as stored procedures or macros. Since iMinMax(θ) employs the classical B^+-tree, it can also be integrated into a DBMS's kernel for greater performance gain.

6 Approximate KNN processing with iMinMax

In high-dimensional databases such as multimedia feature databases, the two most frequently used operations are the similarity range and KNN searches. The similarity range search retrieves objects within a given distance ϵ from a given object, while the KNN search retrieves the K-most similar objects that are closest in distance to a given object. The similarity range search is a spherical range search that can be easily implemented by the range search outlined in the previous section, and hence we shall concentrate on KNN search in this section.

In many high-dimensional applications, small errors can be tolerated. As such, determining approximate answers quickly has become an acceptable alternative. In this section, we propose a novel algorithm that uses the iMinMax(θ) technique to support approximate KNN queries. As in other indexes, it facilitates fast initial response time by providing users with approximate answers online that are progressively refined till more accurate answers are obtained. Eventually, if the user chooses not to terminate prematurely, the precise answers will be obtained. We note that the approximate answers are organized into two categories: those that are certain to be in the final answer set and those without guarantees. It is the latter category that changes as answers are refined.

6.1 Filter-and-refine KNN algorithms

In this paper, we use the Euclidean distance function to determine the distance between two points. Intuitively, a nearest neighbor search can be performed using a filter-and-refine strategy. A range query is used to generate a superset of the answers, and a refinement strategy is then applied to prune away the false drops. We shall illustrate this with an example in two-dimensional space as shown in Fig. 19. Given a search point P, if we want to get two of its nearest neighbors, radius r is needed to have a search region that contains these two points. To check all the points contained in this sphere, a window query with side $2r$ will be conducted on the iMinMax indexing structure. Since the region bounded by the window query is larger than the search sphere, false drops may arise. In our example, points A and C are the nearest neighbors, while point B is the false hit. Clearly, it is almost impossible to determine the optimal range query without additional information. A range query that returns too few records may lead to the wrong answers since a point satisfying a range query need not be a nearest neighbor. A range query that returns too many records will incur too much overhead.

Our solution is to adopt an iterative approach. In the first iteration, a (small) range query (corresponding to d range subqueries on iMinMax) with respect to the data point is generated based on statistical analysis. Some approximate answers can then be obtained from this query. In the second iteration, the query is expanded, and more answers may be returned. This process is repeated until all desired KNN are obtained. We also note that by continuously increasing the search area, the hypersphere based on the Euclidean distance may exceed the data space area. Of course, the area outside the data space has no points to be checked. Figure 20 illustrates such a query.

We note that a naive method for implementing the "query expansion" strategy is to use a larger search window, e.g.,

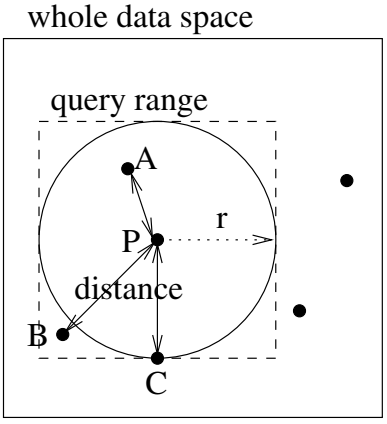


Fig. 19. Query region of a given point

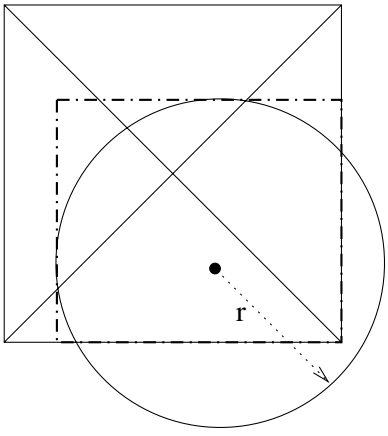


Fig. 20. Enlargement of search region

we apply a sequence of range queries with radii r, $r + \Delta r$, $r + 2\Delta r$, ... until we obtain a value that contains at least K-nearest neighbors. However, this method introduces the additional complexity of determining what Δr should be. More importantly, the search of a subsequent query has to be done from scratch, wasting all effort that was done in previous queries and resulting in very poor performance.

Instead, we adopted a "creeping" search strategy that only incrementally examined the expanded portion of a subsequent query. The idea is to continue the search from where the previous iteration left off by looking at the left and right neighbors. Figure 21 shows pictorially the "creeping search" and its effect on "query expansion". Here we see that, initially, the search region may be within a single partition of the space. However, as the query range expands, more partitions may need to be examined.

Intuitively, the search begins with a small radius that gets increasingly larger until the search radius is max_r. Note that max_r should be set to cover the entire search space to ensure the answers are correct; otherwise, the algorithm may terminate prematurely with badly approximate solutions. However, if all the KNN can be found within a sphere with radius r ($< max_r$), the algorithm will terminate. We note that in each iteration of the proposed scheme, only approximate answers are produced. This is because the data points are indexed based on either the maximum or minimum attribute value, without capturing any similarity information.

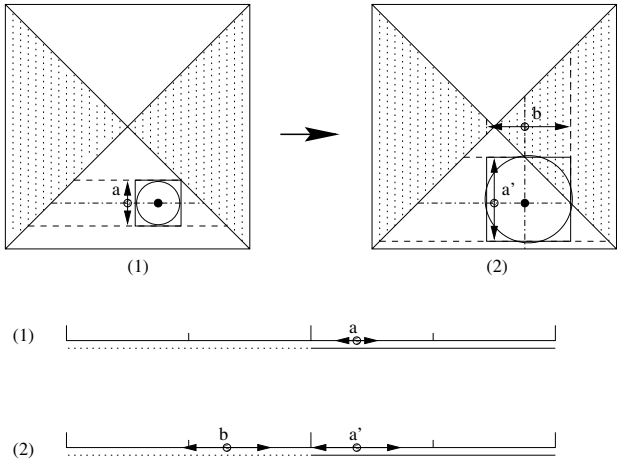


Fig. 21. "Creeping" search on iMinMax index tree

6.2 Quality of KNN answers using iMinMax

Since iMinMax operates with one dimension, it does not capture similarity relationships between data points in high-dimensional space. To study the quality of (approximate) KNN answers returned using iMinMax, we conducted a series of experiments. The quality of answers is measured against the exact KNN. For example, when $K = 10$, if nine out of ten nearest neighbors obtained from iMinMax are in the actual answer set, then we say that the quality of the answer is 0.9. It is important to note that, although the other points are not in the actual answer set, they are close to the nearest neighbor points due to the low contrast between data points with respect to a given point in high-dimensional data space.

In this experiment, we shall examine the effect of the search radius of the window query used by iMinMax(θ) on the accuracy of the KNN returned for that radius. As before, we define accuracy to mean the percentage of KNN in the answer returned by a scheme over the actual KNN. We also examined the KNN returned by an optimal scheme. For the optimal approach, we enlarge the search radius gradually, and for each step we scan the whole database to check for data items that will fall within the search radius. The answer is optimal, as only the nearest neighbors will be included in the answer set.

We used the uniform data set, as it is the data distribution that presents the most difficulty in KNN searches due to low contrast between data points [5]. Figure 22 shows the percentage of KNN data points against that of the optimal approach. As the radius increases, more KNN points are obtained. Surprisingly, the result shows that a small query radius used by iMinMax(θ) is able to get a high percentage of accurate KNNs. Indeed, the results show that the percentage of the K nearest points found is more than that found using the optimal approach with respect to the search radius. This is because, under iMinMax, we have a set of KNNs that we have no guarantee will be in the final answers with the current radius, but these KNNs are actually in the final KNN answer set. This again demonstrates that iMinMax can produce very good quality approximate answers.

6.3 KNN query efficiency

To evaluate the KNN query performance of iMinMax, we implemented two different space partitioning strategies for iDistance as outlined in [17]: a fixed space partitioning strategy for the uniform data set and dynamic-cluster-based partitioning strategy for the skewed data set. For the latter, we implemented two different strategies for the selection of reference points: one uses the centroid of the cluster as the reference point and the other uses the edge point closest to the cluster. Our first experiment uses a 100K 30-dimensional uniform data set, and the query is a 10-nearest neighbor query. Figure 23a shows the result of the experiment. We note that iMinMax can produce quality approximate answers very quickly compared to linear scan. As shown, the I/O cost is lower than linear scan with up to 95% accuracy (here we define accuracy rate as the percentage of answers that are the KNN points). However, since the data are uniformly distributed, all points are almost equidistant to one another. As such, iMinMax ends up scanning almost the entire data set in order to retrieve all 10-nearest neighbors, resulting in a longer total time than linear scan. In fact, uniform data distribution is noted as the most problematic data distribution for KNN queries due to its low contrast. It has been argued that for such a distribution, a KNN query is not meaningful [5]. However, it should be noted that data distributions change and do not conform to some predefined conditions, and hence an index should be built to handle all distributions well. Next, we observe that iMinMax performs as well as iDistance. Since iDistance is a metric-based KNN processing technique specially designed for KNN searches, this result shows the robustness and effectiveness of iMinMax for KNN queries.

In another set of experiments, we use a 100K 30-dimensional clustered data set. The query is still a 10-nearest neighbor query. Figure 23b summarizes the result. The relative performance of the two schemes is similar to that in the case of the uniform data set, i.e., iMinMax remains effective in producing approximate answers quickly (as compared to linear scan). However, we note that it is now less effective compared to iDistance. However, because it is a general and simple structure, we believe iMinMax will be a valuable structure.

6.3.1 On CPU cost

While linear scan incurs less seek time, linear scan of a feature file entails an examination of each data point (feature) and the calculation of distance between each data point and the query point. Further, due to the limited buffer size, the feature file may be scanned intermittenly. The above factors will impact the overall CPU time. Figure 24 shows the CPU time of linear scan, iMinMax(θ), and iDistance for the clustered data set. It is interesting to note that the performance in terms of CPU time approximately reflects the trend in page accesses. The performance of iMinMax(θ) is good when a small amount of error is tolerable. At an accuracy of 90%, the CPU cost of iMinMax(θ) is about half of linear scan's. As the accuracy increases, the CPU cost increases due to the fact that more data points are being examined. The result shows that the iMinMax(θ) is reasonably efficient for approximate KNN search in terms of CPU cost.

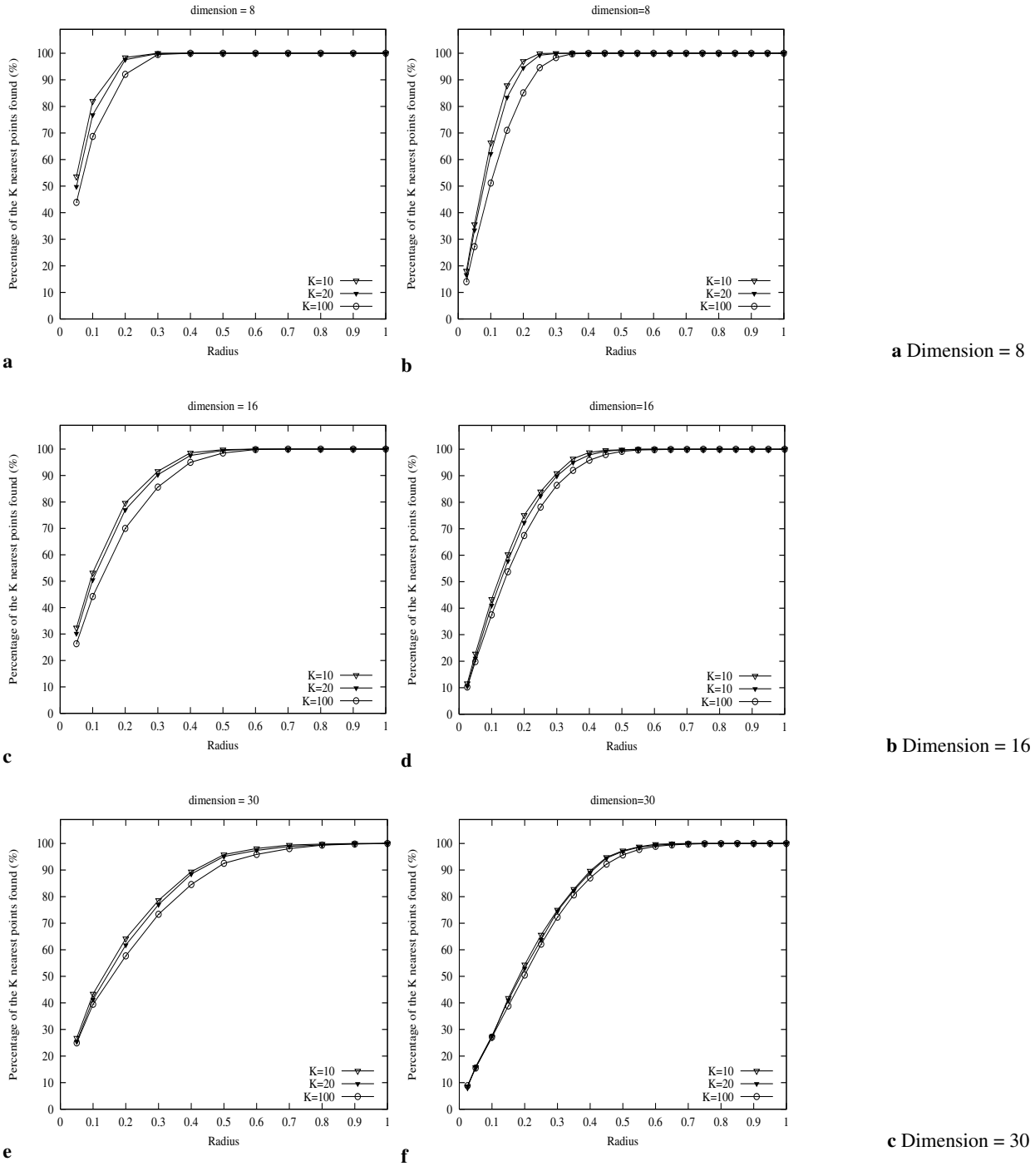


Fig. 22a–f. Probability of finding a certain number of neighbors, with varying searching radius. **a** iMinMax. **b** Optimal. **c** iMinMax. **d** Optimal. **e** iMinMax. **f** Optimal

7 Conclusion

In this paper, we have proposed a simple and yet very efficient method for indexing high-dimensional data based on edges. We have shown by extensive performance studies that the method is more efficient and dynamic than the Pyramid technique and VA-file, as well as linear scan, for window queries. Performance difference is expected to increase as the data volume and dimensionality increase; this should happen with skewed data distributions as well. We have also generalized iMinMax(θ) for nearest neighbor search, and the performance studies show that it is efficient for approximate KNN search when slight inaccuracies can be tolerated. Further, being a B^+-tree-based index, iMinMax(θ) can be crafted easily into existing database backends or implemented as a stored procedure.

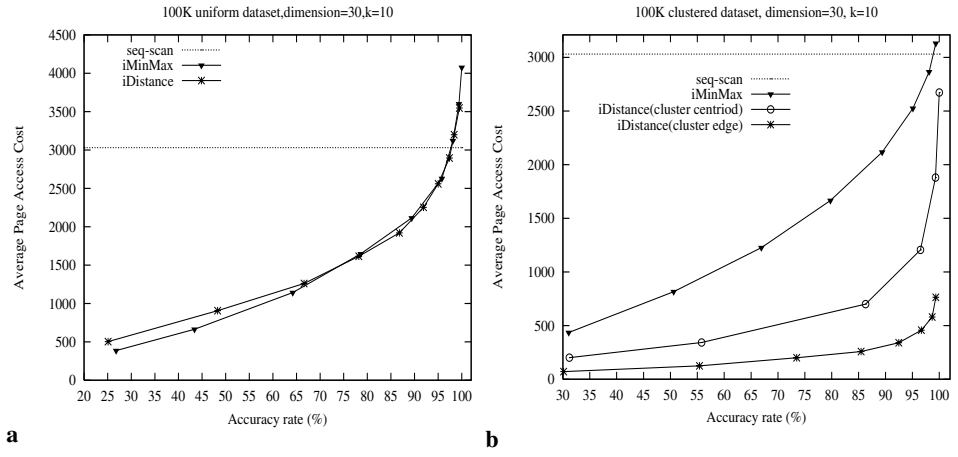


Fig. 23. A comparative study. **a** Uniform data set. **b** Clustered data set

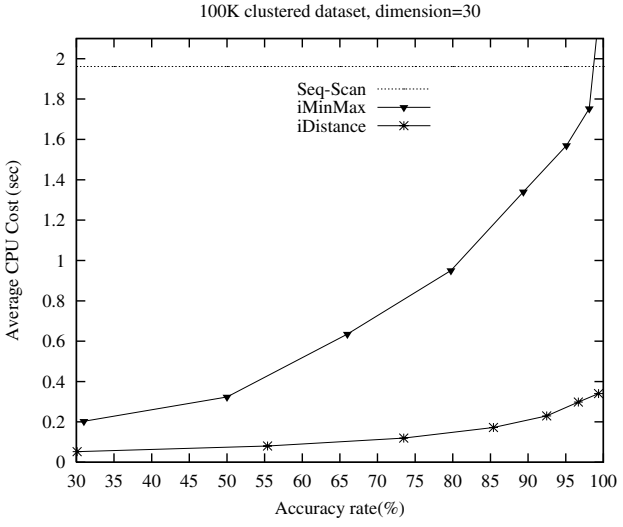


Fig. 24. CPU time

Acknowledgements. This research is part of a system project called VIPER[11], funded by research grant RP 950694. We thank the referees for their useful comments.

References

1. Beckmann N, Kriegel H-P, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD international conference on management of data, Atlantic City, NJ, 23–25 May 1990, pp 322–331
2. Berchtold S, Bϕhm B, Kriegel H-P (1998) The pyramid-technique: towards breaking the curse of dimensionality. In: Proceedings of the 1998 ACM SIGMOD international conference on management of data, Seattle, 2–4 June 1998, pp 142–153
3. Berchtold S, Keim DA, Kriegel H-P (1996) The X-tree: an index structure for high-dimensional data. In: Proceedings of the 22nd international conference on very large data bases, Mumbai (Bombay), India, 3–6 September 1996, pp 28–37
4. Bertino E et al (1997) Indexing techniques for advanced database systems. Kluwer, Dordrecht
5. Beyer K, Goldstein J, Ramakrishnan R, Shaft U (1999) When is nearest neighbors meaningful? In: Proceedings of the international conference on database theory, Jerusalem, Israel, 10–12 January 1999, pp 217–235
6. Böhm C, Berchtold S, Keim D (2001) Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput Surv 33(3):322–373
7. Chan CY, Ooi BC, Lu H (1992) Extensible buffer management of indexes. In: Proceedings of the 18th international conference on very large data bases, Vancouver, BC, Canada, 23–27 August 1992, pp 444–454
8. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on management of data, Boston, 18–21 June 1984, pp 47–57
9. Kamel I, Faloutsos C (1994) Hilbert r-tree: an improved r-tree using fractals. In: Proceedings of the 20th international conference on very large data bases, Santiago de Chile, Chile 12–15 September 1994, pp 500–509
10. Manopopoulos Y, Theodoridis Y, Tsotra VJ (2000) Advanced database indexing. Kluwer, Dordrecht
11. Ooi BC, Tan KL, Chua TS, Hsu W (1992) Fast image retrieval using color-spatial information. J Very Large Databases 7(2):115–128
12. Ooi BC, Tan KL, Yu C, Bressan S (2000) Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In: Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, Dallas, TX, 15–17 May 2000, pp 166–174
13. Ramakrishnan R, Gehrke J (2000) Database management systems. McGraw-Hill, New York
14. Sakurai Y, Yoshikawa M, Uemura S (2000) The a-tree: an index structure for high-dimensional spaces using relative approximation. In: Proceedings of the 26th international conference on very large data bases, Cairo, Egypt, 10–14 September 2000, pp 516–526
15. Weber R, Schek H, Blott S (1998) A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: Proceedings of the 24th international conference on very large data bases, New York, 24–27 August 1998, pp 194–205
16. Yu C (2002) High-dimensional indexing: transformational approaches to high-dimensional range and similarity searches. Lecture notes in computer science, vol 2341. Springer, Berlin Heidelberg New York
17. Yu C, Tan KL, Ooi BC, Jagadish HV (2001) Indexing the distance: an efficient method to knn processing. In: Proceedings of the 27th international conference on very large data bases, Rome, Italy, 11–14 September 2001, pp 421–430