Knowl Inf Syst (2010) 22:1–26
DOI 10.1007/s10115-008-0190-y

REGULAR PAPER

Effectiveness of NAQ-tree as index structure for similarity search in high-dimensional metric space

Ming Zhang · Reda Alhajj

Received: 4 August 2008 / Revised: 12 November 2008 / Accepted: 22 November 2008 /
Published online: 23 January 2009
© Springer-Verlag London Limited 2009

Abstract Similarity search (e.g., k-nearest neighbor search) in high-dimensional metric space is the key operation in many applications, such as multimedia databases, image retrieval and object recognition, among others. The high dimensionality and the huge size of the data set require an index structure to facilitate the search. State-of-the-art index structures are built by partitioning the data set based on distances to certain reference point(s). Using the index, search is confined to a small number of partitions. However, these methods either ignore the property of the data distribution (e.g., VP-tree and its variants) or produce non-disjoint partitions (e.g., M-tree and its variants, DBM-tree); these greatly affect the search efficiency. In this paper, we study the effectiveness of a new index structure, called Nested-Approximate-eQuivalence-class tree (NAQ-tree), which overcomes the above disadvantages. NAQ-tree is constructed by recursively dividing the data set into nested approximate equivalence classes. The conducted analysis and the reported comparative test results demonstrate the effectiveness of NAQ-tree in significantly improving the search efficiency.

Keywords Knn search · High dimensionality · Dimensionality reduction · Indexing · Similarity search

1 Introduction

Similarity search in high-dimensional metric space is the key operation in many applications. It covers two types of queries: range queries and k-nearest neighbor queries (knn search). Range queries may be expressed as follows: given a data set S, a query point q and range r, find all data points $p \in S$ that satisfy $D(p, q) < r$, where $D(p, q)$ gives the distance between p and q. On the other hand, knn search may be specified as follows: given a query point q,

M. Zhang · R. Alhajj (✉)
Department of Computer Science, University of Calgary, Calgary, AB, Canada
e-mail: alhajj@ucalgary.ca

R. Alhajj
Department of Computer Science, Global University, Beirut, Lebanon

find the k nearest points to q; knn search can be regarded as a dynamic range query with r being constantly updated by the distance to the current kth nearest neighbor. A query that only looks for nearest neighbors within certain distance to q is called radius-limited nearest neighbor query.

In this study, we consider metric space, which is a pair (S, D), where S is a data space and D is a distance metric defined on S; such that D satisfies the three properties: (1) $\forall x \in S$, $D(x, x) = 0$; (2) $\forall x, y \in S$, $D(x, y) = D(y, x) \geq 0$; (3) $\forall x, y, z \in S$, $D(x, y) + D(y, z) \geq D(x, z)$. The third property is the metric triangle inequality.

In real world applications: (1) the number of data points is huge and they are usually stored on disk; and (2) the dimensionality of data points is high and the distance computation is expensive. Therefore, exhaustive search is unacceptable. Hence, index structures are needed to prune the search space such that the number of disk accesses and the number or complexity of distance computation can be reduced.

State-of-the-art indexes are constructed based on partitioning of the data set using distances of data points to certain reference points such that the query result falls into a small number of partitions. Most of them follow the two approaches proposed by Burkhard and Keller [7]. One type of methods (including VP-tree and its variants) produce disjoint partitions, but ignore the distribution properties of the data points. The other type of methods (including M-tree and its variants) produce non-disjoint partitions, which greatly affect the search performance.

In this paper, we study the effectiveness of a new index structure, called Nested-Approximate-eQuivalence-class tree (NAQ-tree), that overcomes the above disadvantages. It combines the advantages of the above-mentioned two types of methods. NAQ-tree is constructed by recursively partitioning the data set into nested disjoint approximate equivalence classes. NAQ-tree greatly improves the search efficiency. In addition, NAQ-tree can answer some radius-limited similarity search queries by visiting one leaf node. These properties of NAQ-tree are supported by the conducted analysis and the reported comparative test results. In the experiments, we used three data sets and compared NAQ-tree with state-of-the-art tree structures described in the literature. The data sets used in the experiments are the Corel data set [10], the Phy_train data set [18], and the data set used by the authors of iDistance [16]. The reported results show that NAQ-tree outperforms some of the major index structures described in the literature, include iDistance, VP-tree and DBM-tree. All the reported results support the following argument: with NAQ-tree less nodes are visited, number of disk accesses is lower, and less distance computations are needed; NAQ-tree also demonstrated good scalability.

The rest of the paper is organized as follows. Section 2 presents the related works. Section 3 covers the different aspects of NAQ-tree. Section 4 reports the experimental results. Section 5 is conclusions and future work.

2 Related work

As described in the literature, partitioning methods can be classified into two groups [32]: (1) space-based partitioning methods [22], and (2) data-based partitioning methods. The former methods partition the data space using coordinate planes; thus, the partitions are disjoint hyper-rectangles. Space partitioning methods produce huge number of partitions (exponential in the number of dimensions) and some of the partitions may contain very few points or may be even empty, so that the page storage utilization is very low [17]. Data-based partitioning methods overcome these demerits. R-tree and R^*-tree were originally designed to

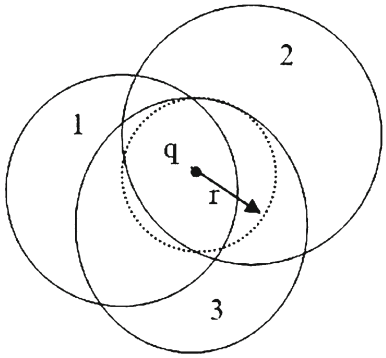


Fig. 1 Effect of the non-disjoint partitions

index spatial objects, hence called spatial access methods (SAM). They produce non-disjoint partitions. SS-tree [31], SR-tree [17], and X-tree [3] can be regarded as variations of the R^*-tree; they try to reduce the overlapping of partitions.

VP-tree [27,33] and its variants, including MVP-tree [5] and dynamic VP-tree [14] follow the first approach of Burkhard and Keller [7], and produce disjoint partitions. The original VP-tree [27] selects the reference points randomly. Then, Yianilos [33] proposed to use sampling techniques to select reference point. MVP-tree [5] stores the distances from each data point to the reference points and uses them to avoid unnecessary distance computations in the query search process. Further, MVP-tree was designed to reduce distance computations for range queries only, not for actual knn queries. VP-tree and MVP-tree are both static, no insert operations were introduced. Later, Fu et al. [14] proposed the insert and delete operations for the VP-tree. Yianilos [34] proposed VP-forest (which is a set of modified VP-trees) for radius-limited nearest neighbor search. The D-index proposed by Dohnal et al. [12] employs an idea similar to the VP-forest. However, both VP-forest and D-index do not give any performance guarantee for general nearest neighbor queries.

Gh-tree [27], GNAT [6], M-tree [9], Slim-tree [26], DBM-tree [29], and Δ^+-tree [11] follow the second approach of Burkhard and Keller [7]; they produce non-disjoint partitions. These methods can be regarded as clustering-based methods, and none of them can completely remove the overlapping. Other recent works that worth mentioning are: Omni [13], iDistance [16] and the method developed by Venkateswaran et al. [28].

Generally, the case of disjoint partitions is better than non-disjoint partitions in terms of search efficiency. In the case of disjoint partitioning, the query point can fall in only one partition. When the query range r is small enough, the whole query hyper-sphere is in one partition. As we only need to search the partitions that intersect the query hyper-sphere, we can prune all the other partitions. On the other hand, in the case of non-disjoint partitioning (as shown in Fig. 1), if the query point q falls in the overlapped area, we need to search all the three partitions no matter how small the range r is. However, it is worth noting that the existing disjoint partitioning methods (VP-tree and its variants) are not always better than non-disjoint partitioning methods. The former type of methods is implicitly based on the assumption that data points are uniformly distributed, which usually does not hold in real-world applications. In order to keep the tree balanced, they divide the data set into equal-sized partitions, ignoring the inherent grouping of data points. On the contrary, the non-disjoint partitioning methods (M-tree and its variants) capture this property by dividing the data set by clustering. For example, in situations similar to the case illustrated in Fig. 2,

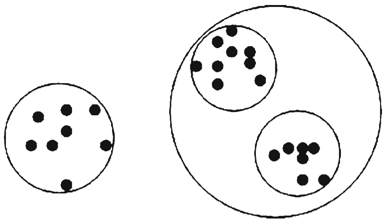


Fig. 2 Sparsely distributed clusters of data points

where the data point clusters are sparsely distributed, the clustering-based methods are better as they may produce tighter partitions and improve the pruning rate.

To sum up, each of the two partitioning approaches described above has its advantages. Fortunately, NAQ-tree is an index structure that combines the advantages of the two types of methods (disjoint partitioning and non-disjoint partitioning). In NAQ-tree, a data set is divided into disjoint partitions by respecting the data distribution. The test results reported in this paper demonstrate the power of the NAQ-tree over the well-known and heavily cited representatives of the other two types of approaches.

We have discussed the existing index structures for exact similarity search in generic metric space. However, some index structures are designed for specific applications, e.g., the Compact Multi-Resolution Index for time series databases [20]. Other techniques for similarity search include: the Locality Sensitive Hashing scheme [2], which has been designed for approximate nearest neighbor search; VA file [32] which uses vector approximation to accelerates sequential scan; CVA [1] file which improves the VA file by incorporating critical-value based dimension-reduction technique; the multi-step method [24] which uses filter-refinement strategy to confine expensive distance computation in small filtered candidate set. Finally, Kailing [19] combines the multi-step method with the metric index structure for range queries of complex objects.

3 NAQ-tree

Consider a set of objects $O = \{o_1, o_2, \ldots, o_n\}$ and a set of attributes $A = \{a_1, a_2, \ldots, a_d\}$, we first divide the objects into groups based on the first attribute a_1, i.e., objects with same value of a_1 are put in the same group; each group is an *equivalence class* [23] with respect to a_1. In other words, all objects in a group are indistinguishable by attribute a_1. We can refine the equivalence classes further by dividing each existing equivalence class into groups based on the second attribute a_2; all objects in a refined equivalence class are indistinguishable by attributes a_1 and a_2. This process may be repeated by adding one more attribute at a time until all the attributes are considered. Finally, we get a hierarchical set of equivalence classes, i.e., a hierarchical partitioning of the objects. This is roughly the basic idea of NAQ-tree, i.e., to partition the data space in our similarity search method. In other words, given a query object o, we can gradually reduce the search space by gradually considering the most relevant attributes.

In our similarity search method, each object is a d-dimensional data point $P = (p_1, p_2, \ldots, p_d)$, $p_j \in R$; attributes are the distances from data points to reference points (selected using sampling-based method as described in [33]). At each level, we compute the distances from the data points to a reference point. Each data point is represented by its distance to the reference point. This is equivalent to mapping the data points into a one-dimensional distance space (distances are scalar). This way, we cluster the data points

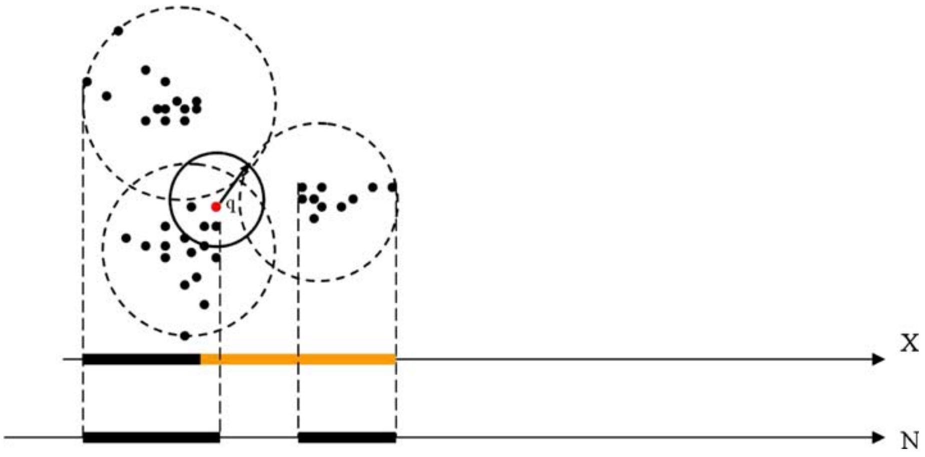


Fig. 3 Example data points with different perspectives

in this one-dimensional space. Each cluster is a partition, called an *approximate equivalence class*, in the sense that the data points in the same class have similar distances to the reference point, and are hardly distinguishable by this attribute. We apply this strategy recursively in the partitions until each partition is small enough to fit into one disk page. Finally, we get a set of nested approximate equivalence classes, which is a hierarchical disjoint partitioning of the data set.

As we partition the data points based on the distances to selected reference points, desirable reference points should give large variance of distances so as to better separate the data points. Ideally, the selected reference point is expected to maximize the variance. However, determining an "ideal" reference point is too expensive. Thus, we employ the sampling-based method [33]: randomly select two sets of sampling data points, denoted by A and B; A works as the candidate set of reference points; compute the distances from each points in A to the points in B; the point (from A) with the largest variance of distances (to points in B) is selected as the reference point. The conducted experiments show that the sampling-based method works almost as well as selecting "ideal" reference point. More importantly, the search performance is stable when repeating the experiments.

To explicitly highlight the novelty of NAQ-tree, herein we elaborate further on how NAQ-tree benefits from and combines the advantages of both disjoint-partitioning methods and clustering-based methods. NAQ-tree partitions the data set based on the distances to one reference point. As there exists a full order on the distances to the same reference point, the partitions are disjoint. In this respect, NAQ-tree is like the disjoint-partitioning methods (VP-tree and its variants). On the other hand, NAQ-tree does not partition the data set evenly. It clusters the data points in one-dimensional space (based on distance, which is scalar); each cluster is a partition. This partitioning strategy captures the data distribution, in the sense that far away data points are separated into different partitions. In other words, data points are classified into their natural groups. In this respect, NAQ-tree is similar to the clustering-based method (M-tree and its variants). Therefore, NAQ-tree combines the advantages of the two types of existing methods discussed in Sect. 2.

To illustrate the power of NAQ-tree, lets look at how NAQ-tree and the two types of methods described in Sect. 2 handle the data set and the query shown in Fig. 3. Clustering-based methods (M-tree and its variants) cluster the data set shown in Fig. 3 into three groups and use a bounding sphere (the dotted circle) to represent each cluster, usually these bounding spheres do overlap. Given a query sphere as shown in Fig. 3, it intersects all the three bounding spheres, thus requires searching all the clusters. Existing disjoint-partitioning methods (VP-tree and its variants) map the data points to the X-axis based on their distance to a reference point, and divide the points evenly into same-sized partitions. As illustrated in Fig. 3, the two partitions are not separated and the partition boundary is in the denser area. When mapped to the X-axis, the query sphere intersects both data partitions, thus it is required to search both. NAQ-tree maps the data points to the N-axis and clusters the points into two groups; the two clusters are separated by the natural space between them. When mapped to the N-axis, the query sphere only intersects one of the partitions, thus requires searching only one partition.

At the end, it is worth mentioning that as we do not partition the data evenly, NAQ-tree is unbalanced, which may be considered as another advantage of NAQ-tree. We show next in Proposition 3.1 that for similarity search in high dimensional space, unbalanced tree may provide better performance than balanced tree. Further, Proposition 3.2 shows that having a balanced tree does not provide any advantage for uniformly distributed data in general.

Proposition 3.1 *For similarity search in high dimensional space, unbalanced tree may provide better performance than balanced tree.*

Proof We know that, if there exists a full order on the search key, then the exact search follows a single root-to-leaf path, and range search looks into extra consecutive leaf nodes. In this case, the search cost is determined by the tree height and a balanced tree minimizes the height. However, for similarity search in high dimensional space, there does not exist a full order on the search key; so the search has to follow many branches. The number of branches to be searched is determined by the search radius and the data distribution. That is, the search cost is determined by the pruning rate of the search space, not by the tree height. The pruning rate of the search space is determined by how the data set is separated. The balanced tree partitions the data set into equal-sized parts, ignoring the data distribution. NAQ-tree partitions the data set by the data distribution; thus it is better separating the data set than balanced partitioning. For more detailed justification of the benefits of unbalanced tree in similarity search, refer to [8]. □

Proposition 3.2 *The balance of a tree does not provide any advantage for uniformly distributed data.*

Proof Beyer et al. [4] proved that under certain broad conditions of the data and query distributions (uniform distribution is one of them), nearest neighbor query becomes unstable with the increase of dimensionality. Specifically, as the dimensionality increases, the difference between the distance to the nearest neighbor and the distance to the farthest neighbor does not grow as fast as the distance to the nearest neighbor. That is, the contrast between the distance to the nearest neighbor and the other points is diminishing. Shaft and Ramakrishnan [25] extended the result of Beyer et al. [4] by proving that the expected performance of any Convex Description Index structure converges to the performance of linear scan as the dimensionality increases. Convex Description Index includes a large group of existing indexes, e.g., R-tree and its variants, M-tree and its variants, etc. The result of these two theoretic works shows that we cannot expect an index structure to work well for uniformly distributed data in high dimensional space. Therefore, the balance of tree does not provide any advantages for uniformly distributed data. □

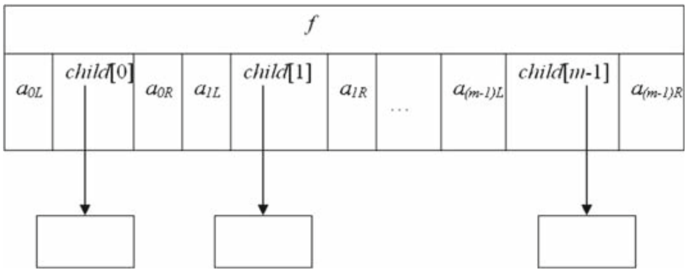


Fig. 4 Internal node structure in NAQ-tree

Algorithm 1 *Tree-construct*(U, d, N)

```
1: /*$U$ is a pointer to the $d$-dimensional data set with $N$ data points*/
2: $Root = Node\text{-}construct(U, d, N)$; /*invoke Algorithm 2.*/
```

3.1 Tree construction

As described in Definition 3.1, NAQ-tree is an unbalanced index structure representing the hierarchical disjoint partitions of the data set. Leaf nodes of the tree store data points and non-leaf (internal) nodes contain the partition information. In the remaining part of the paper, we may interchangeably use the terms non-leaf and internal node.

Definition 3.1 (Characteristics of NAQ-tree) NAQ-tree is an unbalanced tree with these properties:

1. Only leaf nodes store data points;
2. For any non-leaf node X, all the data points in its descendant leaf nodes are said to be covered by X;
3. Each non-leaf node X records:
   (a) One reference point f;
   (b) m pairs of partition boundaries $[a_{iL}, a_{iR}]$ $(i = 0..(m-1))$ of the data points covered by X, where the partitions are obtained based on the distances from the data points to reference point f;
   (c) m pointers child$[i]$ $(i = 0..(m-1))$ to the child nodes, where child$[k]$ $(0 \leq k < m)$ points to the child node that covers the data points in $[a_{kL}, a_{kR}]$. □

An internal node in the NAQ-tree looks as shown in Fig. 4.

The tree construction process is performed in a top-down fashion starting from the root node, which covers all the data points. We select a reference point (using sampling-based method as described in [33]) for the root node, compute the distances from all the data points to the selected reference point and cluster the data points based on these distances. Each cluster (partition) is covered by one child node of the root. This process is carried out recursively in the child nodes until each partition can fit into a disk page; this requires invoking Algorithm 1 that calls Algorithm 2 to complete the tree construction process.

In Algorithm 2, as each data point is represented by its distance to the reference point, the clustering is performed in one-dimensional space. We consider the data set as a degenerated weighted tree with each data point being a node, and edge weight is the distance between neighboring data points in one-dimensional space. Thus, clustering is performed simply by breaking the $(m-1)$ highest weighted edges. As a result, each set of linked data points form

Algorithm 2 *Node-construct*(U, d, n)

```
1: /*$U$ is a pointer to the $d$-dimensional data set with $n$ data points, $C$ is the leaf node capacity*/
2: if $n < C$ then
3:   /*this is leaf node*/
4:   Node=Initialize_leafnode();
5:   Insert all points in Node;
6:   Return address of Node;
7: else
8:   /*this is internal node*/
9:   find the reference point $f$;
10:  compute the distance from each data point to $f$; /*complexity $O(dn)$*/
11:  sort the data points based on the distances; /*complexity $O(nlogn)$*/
12:  cluster the data points, clusters' boundaries are $[a_{iL}, a_{iR}]$, $i = 0, \ldots, (m-1)$; /*invoke Algorithm 3.*/
13:  Node = Initialize_node();
14:  Node.reference=$f$;
15:  for each cluster with boundary $[a_{iL}, a_{iR}]$, $i = 0, \ldots, (m-1)$ do
16:    Node.left_bound[$i$]= $a_{iL}$;
17:    Node.right_bound[$i$]= $a_{iR}$;
18:    Node.child[$i$]=Node_construct($U_i$,$d$, $n_i$); /*$U_i$ is the pointer to the $n_i$ data points in cluster $i$ */;
19:  end for
20:  Return the address of Node;
21: end if
```

Algorithm 3

```
1: Compute the distances (edge weights) between the neighboring data points;
2: Sort the edges in descending order of weights;
3: do
4: /*check the edges in descending order of weights*/
5: if by breaking the current edge, the resulting two clusters are both larger than the minimum page size then
6:   break the edge;
7: else
8:   move to the next edge;
9: end if
10: while (less than $m-1$ edges have been broken);
```

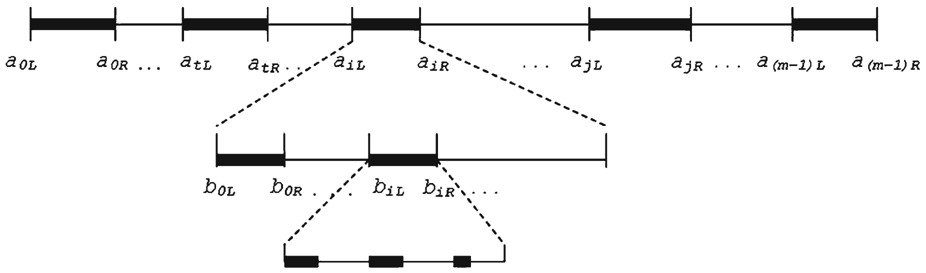


Fig. 5 The hierarchical disjoint partitions

a cluster. The clustering process is performed by invoking Algorithm 3. The **if** condition (line 5) in Algorithm 3 guarantees the minimum space utilization of leaf nodes.

After the tree is constructed, we get hierarchical disjoint partitions of the data set as shown in Fig. 5. At each internal node, the data points are divided into disjoint partitions by their distances to the reference point. In the next section, we will show how the NAQ-tree (constructed as described in this section) helps in similarity search.

3.2 Similarity search algorithms

Given a range query (q, r), where q is the query point and r is the range, it is required to find all points s that satisfy $D(s, q) < r$, where $D(s, q)$ is a metric that gives the distance between s and q. First, we compute the distance $D(q, f)$ between q and the selected reference point f. If $a_{iL} \leq D(q, f) \leq a_{iR}$, i.e., q falls into the partition $[a_{iL}, a_{iR}]$, we first search (using depth-first search) the partition $[a_{iL}, a_{iR}]$; in case q falls in the gap between two partitions, this step can be skipped; and then search (again using depth-first search) the partitions in the left and right directions. In both directions, we compare $D(q, f)$ with the partition boundaries; in the left direction we compare with the right boundaries, and in the right direction we compare with the left boundaries. We can prove that we only need to search a small set of consecutive partitions. Specifically, the search can halt if the condition in Proposition 3.3 is satisfied.

Proposition 3.3 (Stop searching condition:)

1. *In the right direction, if for certain a_{jL}, we have $a_{jL} - D(q, f) > r$, then we do not need to search partitions to the right of a_{jL}; i.e., we can prune partition $[a_{jL}, a_{jR}]$ and all the partitions to the right of it.*
2. *In the left direction, if for certain a_{tR}, we have $D(q, f) - a_{tR} > r$, then we do not need to search partitions to the left of a_{tR}; i.e., we can prune partition $[a_{tL}, a_{tR}]$ and all the partitions to the left of it.*

Proof In the right direction, suppose we have $a_{jL} - D(q, f) > r$, choose an arbitrary data point s that falls to the right of a_{jL}, i.e., $D(s, f) > a_{jL}$.

Since D is a metric, we have the inequality:

$$D(s, q) + D(q, f) \geq D(s, f)$$

That is, $D(s, q) \geq D(s, f) - D(q, f) > a_{jL} - D(q, f) > r$

Since s is arbitrarily chosen, we know that all the data points to the right of a_{jL} are far away from q than the query range r, i.e., fall outside the query range.

By the same way, we can prove the stop condition for the left direction. □

The above description shows that, at each level, we can restrict the search space to a set of consecutive partitions (Approximate eQuivalence classes) based on the distance to one reference point. Within each partition, we can further reduce the search space based on the distances to another reference point.

The knn query is a dynamic range query with r being the distance of the current kth nearest neighbor; at the beginning, r is set to infinity. During the search process, r is updated (decreased) when a new nearest neighbor is found. From the stop condition in Proposition 3.3, we know that the smaller the r is, the smaller the search space will be.

Algorithms 4 and 5 are invoked for performing range queries and knn search, respectively. Algorithm 4 answers the range query: *given query point q and query range r, look for all data points p that satisfy $Dist(p, q) \leq r$, where $Dist(p, q)$ denotes the distance between p and q*. Algorithm 5 answers the knn query: *given query point q, find the k-nearest points to q*. In Algorithms 4 and 5, m is the number of partitions (in one internal node).

Note that for range queries, the search order (of partitions) does not affect the search efficiency; we may search in either direction first. In our algorithm, we search all the partitions to the left of query point q before searching the partitions to the right of q. However, for knn queries, the search order is important. The chances that nearest neighbors exist are higher in the partitions closer to q than in the partitions far away from q; hence, the best search order

Algorithm 4 *Search_rq*$(q, r, Node)$

```
1: /*Initially $Node$=$Root$*/
2: Compute $D_q = Dist(q, Node.reference)$;
3: if $Node$ is not a leaf node then
4:   if $q$ falls in a partition $x$ then
5:     $Search\_rq(q, r, Node.child[x])$;
6:   end if
7:   Let $i$ and $j$ be, respectively, the left and right partitions adjacent to $q$;
8:   while $i \geq 0$ and $D_q - Node.right\_bound[i] \leq r$ do
9:     /*search the left direction*/
10:    $Search\_rq(q, r, Node.child[i])$;
11:    $i = i - 1$;
12:  end while
13:  while $j \leq (m - 1)$ and $Node.left\_bound[j] - D_q \leq r$ do
14:    /*search the right direction*/
15:    $Search\_rq(q, r, Node.child[j])$;
16:    $j = j + 1$;
17:  end while
18: else
19:   Search the leaf node, add all the data points $p$ that satisfy $Dist(q, p) \leq r$ into the result set;
20: end if
```

Algorithm 5 *Search_KNN*$(q, D_k, Node)$

```
1: /*$D_k$ is the current $k^{th}$ nearest distance to $q$, initially $D_k = \infty$*/
2: Compute $D_q = Dist(q, Node.reference)$;
3: if Node is not a leaf node then
4:   if $q$ falls in a partition $x$ then
5:     $D_k = Search\_KNN(q, D_k, Node.child[x])$;
6:   end if
7:   Let $i$ and $j$ be the left and right partitions adjacent to $q$, respectively;
8:   while $i \geq 0$ or $j \leq (m - 1)$ do
9:     if $i \geq 0$ and $D_q - Node.right\_bound[i] \leq D_k$ then
10:      /*search in the left direction*/
11:      $D_k = Search\_KNN(q, D_k, Node.child[i])$;
12:      $i = i - 1$;
13:    end if
14:    if $j \leq (m - 1)$ and $Node.left\_bound[j] - D_q \leq D_k$ then
15:      /*search in the right direction*/
16:      $D_k = Search\_KNN(q, D_k, Node.child[j])$;
17:      $j = j + 1$;
18:    end if
19:  end while
20:  Return $D_k$;
21: else
22:   Search the leaf node sequentially, update the result set and $D_k$ each time a point $q$ is found that satisfies
      $Dist(q, p) \leq D_k$ ;
23:   Return $D_k$;
24: end if
```

is from close to far with respect to q. In our implementation, we iteratively search one left partition followed by one right partition, starting from the partitions close to q.

Another advantage of NAQ-tree is that it can perform some limited-radius similarity search very efficiently. Specifically, NAQ-tree can give the answer by visiting one leaf node as shown next in Proposition 3.4.

Proposition 3.4 *Let s denotes the minimum length of the gaps between partitions at all levels, for any $r < \frac{s}{2}$, for range query (q, r) and nearest neighbor query within radius r, NAQ-tree can give the answer by visiting at most one leaf node.*

Proof As s is the minimum length of the gaps between partitions, any two neighboring partitions are at least s apart. With $r < \frac{s}{2}$, at each level, the query range (q, r) could intersect at most one partition, thus only needs to follow one path. If at certain level, the query range does not intersect any partition, it can report an empty answer set. For a nearest neighbor query within radius r, the radius r is non-increasing in the search process; so the above description still holds. □

For radius-limited nearest neighbor queries within radius $r < \frac{s}{2}$, we invoke Algorithm 5 with the initial radius set to r, and it can give the answer by visiting at most one leaf node. VP-tree and its variants do not satisfy this property because neighboring partitions are not separated by a gap. Also, non-disjoint partitioning methods (M-tree and its variants) do not have this property because of the overlapping between the partitions. In the test results reported in Sect. 4.2, we will show that the B^+-tree based iDistance method does not satisfy the property stated in Proposition 3.4. Note that when $r = 0$, the radius limited nearest neighbor query becomes exact match query. The iDistance method requires visiting a large number of leaf nodes even for exact match queries.

4 Experiment

To demonstrate the performance and effectiveness of the proposed NAQ-tree, we conducted experiments to compare NAQ-tree with (1) VP-tree as a disjoint partition method; (2) DBM-tree, which may be considered as the best known non-disjoint partition method, and in [29] it has been shown to perform better than M-tree and Slim-tree; and (3) iDistance, which is the latest B^+-tree based similarity search index, and in [16] it has been shown to perform better than both Omni method and M-tree.

We decided on using VP-tree instead of MVP-tree in the comparison because the only advantage of MVP-tree over VP-tree is that MVP-tree stores (in the leaf nodes) the distances of each data point to the reference points and uses these distances to reduce the number of distance computations. This technique has nothing to do with data partitioning; in general, it does not reduce the number of disk accesses. Further, MVP-tree [5] did not report any performance improvement over VP-tree in terms of disk accesses. MVP-tree technique to reduce the number of distance computations can also be adapted to the NAQ-tree. It is left as future work because in this paper we want to concentrate more on showing how our new data partitioning method improves the performance in terms of disk accesses. Actually, the test results reported in this paper reflect indirect comparison of NAQ-tree and MVP-tree in terms of distance computations because we report the improvement that NAQ-tree achieves in terms of distance computations over VP-tree; this is a good indicator of how comparable NAQ-tree and MVP-tree are when we compare the improvement each achieves compared to VP-tree.

We conducted the experiments using two real data sets: (1) Corel Image Features [10], which has 68,040 32-dimensions data points, and (2) the Phy_train [18], which has 50,000 78-dimensional data points. For each data set, a query set of 500 points are randomly selected from the data set. The query set is divided into two parts, each contains 250 points. We removed the second part from the data set. Thus, half of the query set is in the data set and the other half is not. In the remaining part of this section, if not indicated otherwise, all the

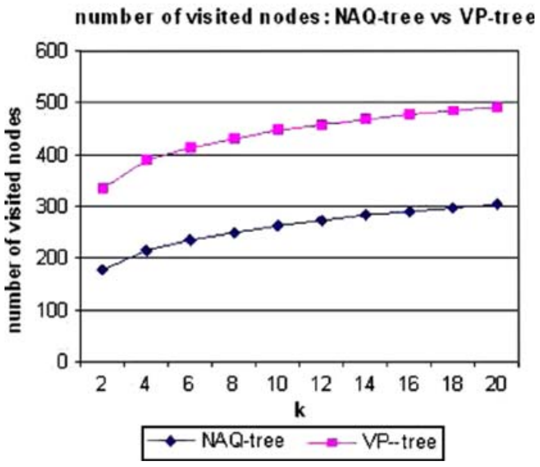


Fig. 6 Comparison of visited nodes using Corel data set

performance measurements represent the average over the 500 queries. All the experiments were performed on a computer with Intel Core(2) 2.4 GHz CPU and 3GB RAM running Kubuntu Linux 7.04.

4.1 NAQ-tree versus VP-tree

We used the VP-tree implementation (special visiting order) available at [30]. In order to make a fair comparison, we set the parameters of NAQ-tree to be the same as VP-tree: (1) tree fanouts is 10; (2) leaf node capacity is 100 points; (3) both methods select the reference points by sampling with the same sampling rate. Further, for knn queries, we tested the performance from $k = 2$ to $k = 20$. We choose this range because most real applications look for nearest neighbors within the range [2, 20]. For example, in the work of Lowe [21], two nearest neighbors of the query feature vector are found in the database, the second nearest neighbor is used to verify if the most nearest neighbor is distinctive.

As we know, the number of disk accesses (per query) is determined by (but not necessarily equal to) the number of visited nodes (per query). In the worst case that each node resides in its own page, the number of disk accesses equals to the number of visited nodes. Figures 6 and 7 show the number of visited nodes for knn queries using Corel (32 dimensions) and Phy_train (78 dimensions), respectively. For Corel, NAQ-tree saved 38–46% of the node visits; and for Phy_train, NAQ-tree saved 32–41% of node visits. This confirms Proposition 3.1 that the unbalanced tree (e.g., NAQ-tree) may outperform the balanced tree (VP-tree) because NAQ-tree separates the data better, and hence the pruning rate is improved.

In both VP-tree and NAQ-tree, internal nodes are much smaller than leaf nodes. In our parameter setting, the size of an internal node is less than 1/50 of the size of a leaf node. Further, the size of the internal node is fixed (for fixed fanouts). In most cases, many internal nodes can be kept in one page, and in some cases all internal nodes can fit into main memory because the total number of internal nodes is much smaller than the number of leaf nodes. Therefore, the number of disk accesses (per query) can be estimated by the number of visited leaf nodes (per query). In this case, the performance improvement of NAQ-tree is even better. Figures 8 and 9 give the number of disk accesses (visited leaf nodes) for knn queries using the two data sets, respectively. For Corel, NAQ-tree reduced 53–65% of the disk accesses; and for Phy_train, NAQ-tree saved 42–52% of the disk accesses.

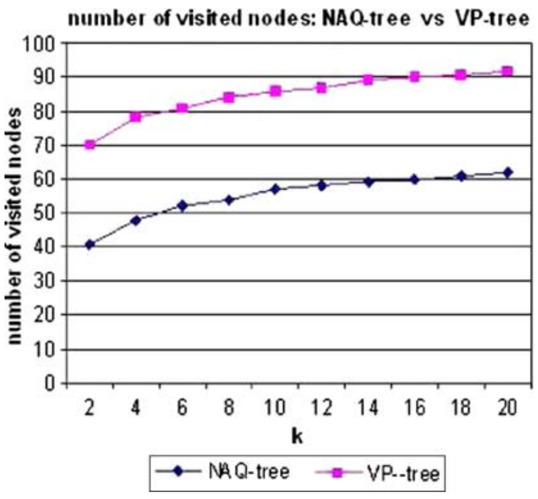


Fig. 7 Comparison of visited nodes using Phy_train data set

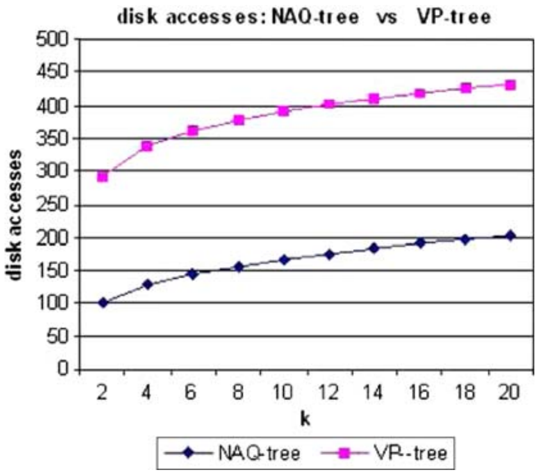


Fig. 8 Comparison of disk accesses using Corel data set

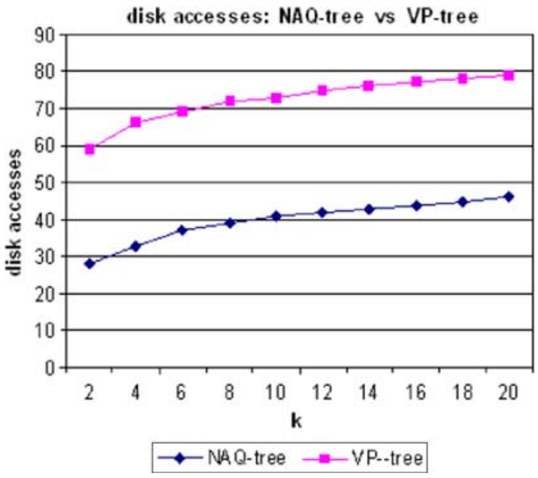


Fig. 9 Comparison of disk accesses using Phy_train data set

Fig. 10 Comparison of distance computations using Corel data set

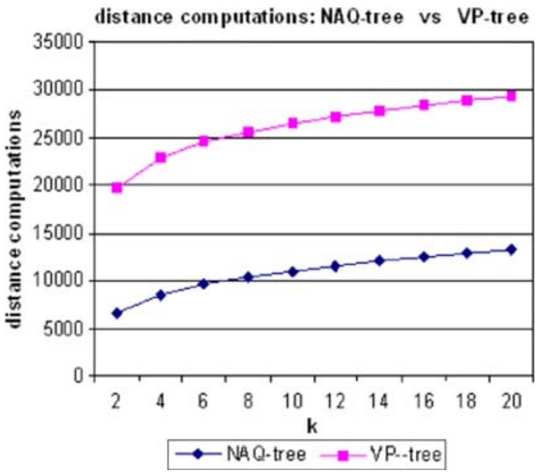
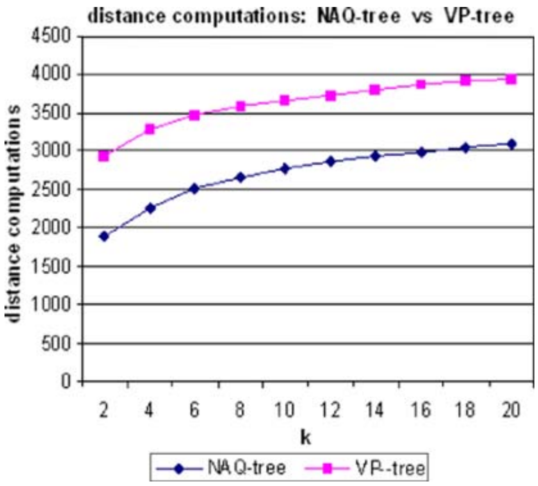


Fig. 11 Comparison of distance computations using Phy_train data set



The number of distance computations is equal to the number of visited data points plus the number of visited reference points. Figures 10 and 11 display the number of distance computations for knn queries using the two data sets Corel and Phy_train, respectively. For Corel, NAQ-tree saved 55–66% of distance computations; and for Phy_train, NAQ-tree saved 22–35% of the distance computations.

For range queries, we tested the range from 0.01 to 0.08 for Corel. We chose this range because there is no point found when the range is expanded below 0.01. When the range $r = 0.08$, the average number of points in the answer set is approximately 22. For Phy_train, we tested the range from 0.6 to 1.2 as there is no point found when the range is below 0.6; and when the range reaches 1.2, the average number of points in the answer set is over 20.

Figures 12 and 13 illustrate the number of visited nodes for the two data sets. Figure 12 shows that, for Corel, when the range is 0.01, VP-tree visits a little less nodes than NAQ-tree, which is due to the fact that NAQ-tree is unbalanced, so it visits more internal nodes than leaf nodes when the range is very small. For range $r \geq 0.02$, on the other hand, NAQ-tree visits less nodes than VP-tree and the performance difference becomes larger as the range

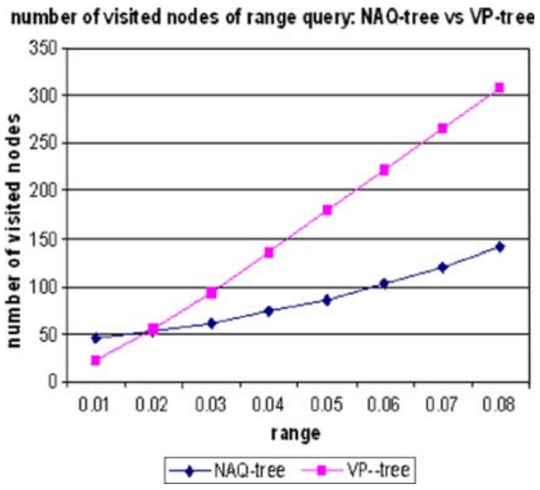


Fig. 12 Comparison of visited nodes using Corel data set

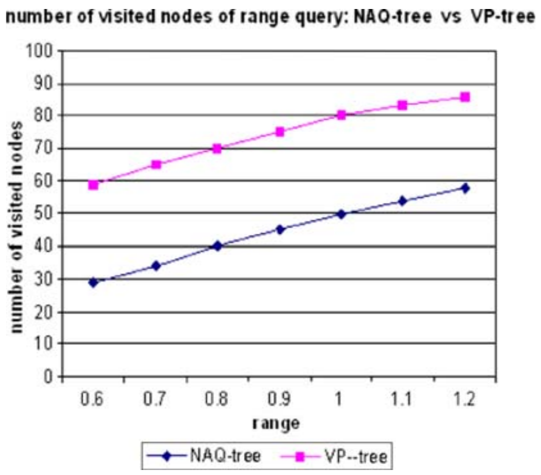


Fig. 13 Comparison of visited nodes using Phy_train data set

increases. As illustrated in Fig. 12, NAQ-tree can save up to 54% of the node visits. For Phy_train, NAQ-tree saved 32–51% of the node visits.

Recall that the number of internal nodes is small and only a leaf node occupies one disk page, hence we can estimate disk accesses by the number of visited leaf nodes. Figures 14 and 15 give the number of disk accesses for the two data sets Corel and Phy_train, respectively, where it can be easily seen that NAQ-tree reduced 74–80% of disk accesses for Corel; and NAQ-tree saved 43–62% of disk accesses for Phy_train. Figures 16 and 17 illustrate the number of distance computations, which demonstrate that NAQ-tree can save up to 80% of the distance computations for Corel, and saved up to 49% for Phy_train.

4.2 NAQ-tree versus iDistance

The iDistance implementation has been obtained from the author's home page at [15]. We set the node capacity of NAQ-tree to be the same as the capacity used for iDistance. And the minimum space utilization of leaf node is set to 50%, which is the same as iDistance (50%

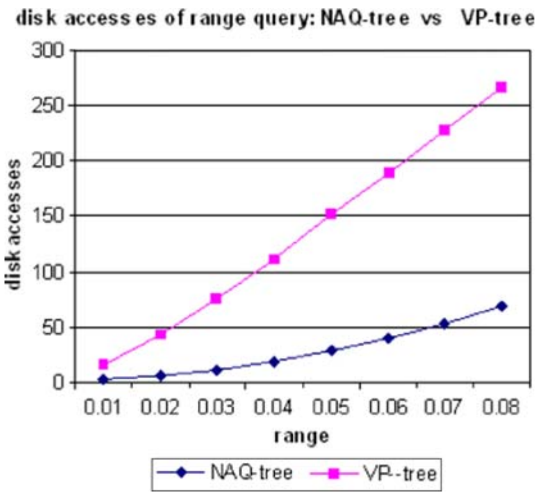


Fig. 14 Comparison of disk accesses using Corel data set

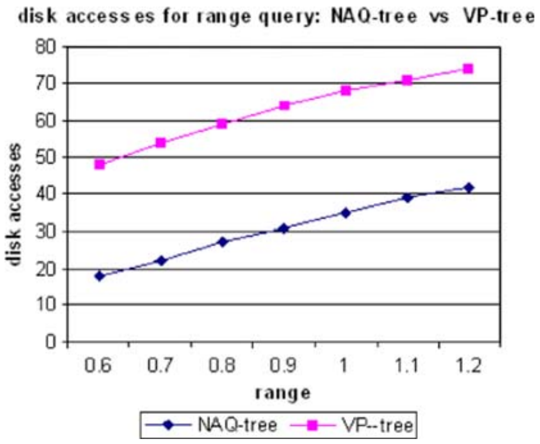


Fig. 15 Comparison of disk accesses using Phy_train data set

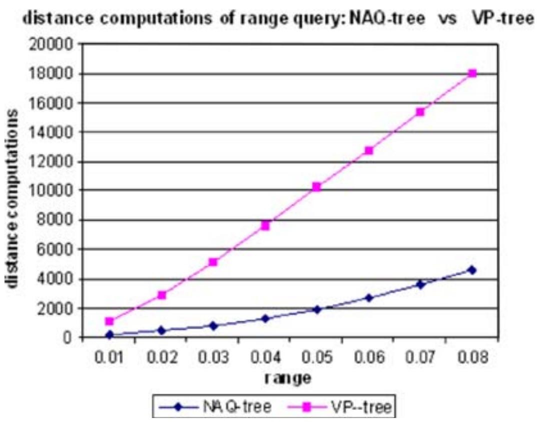


Fig. 16 Comparison of distance computations using Corel data set

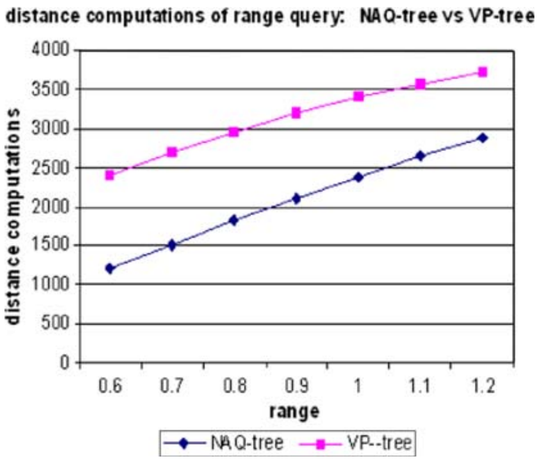


Fig. 17 Comparison of distance computations using Phy_train data set

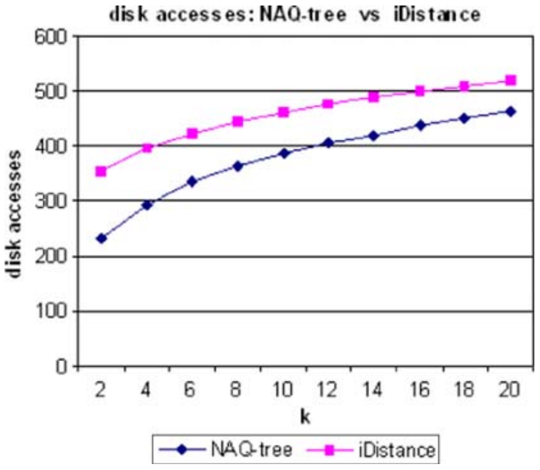


Fig. 18 Comparison of disk accesses using Corel data set

is the guaranteed space utilization rate of B$^+$-tree). The other parameters of iDistance (i.e., the number of reference points, initial search radius, and radius increment at each step) are set to the same values as those reported in [16]. In iDistance implementation, it is assumed that all internal nodes can be placed in main memory so that the number of the disk accesses is the number of leaf nodes visited.

Here, we run the experiments for three data sets. In addition to Corel and phy_train, we also tested using the data set (together with the query set) obtained from the iDistance author's home page [15]. Hereafter, the third data set will be referred to as the iDistance data set.

Figures 18 and 19 show the number of disk accesses for Corel and Phy_train, respectively. These figures show that NAQ-tree saved 10–35% of the disk accesses for Corel, and saved 46–57% of the disk accesses for Phy_train.

Figure 20 illustrates the disk accesses for the iDistance data set, which shows that NAQ-tree can save 18–48% of disk accesses.

Although not described in the original paper, the iDistance implementation provides an option to post-process the B$^+$-tree to compact the leaf nodes so that the space utilization of leaf nodes can be improved to a user defined rate. Figure 21 shows the disk accesses after

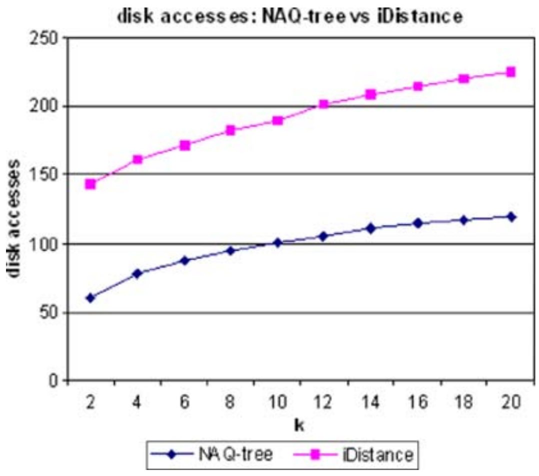


Fig. 19 Comparison of disk accesses using Phy_train data set

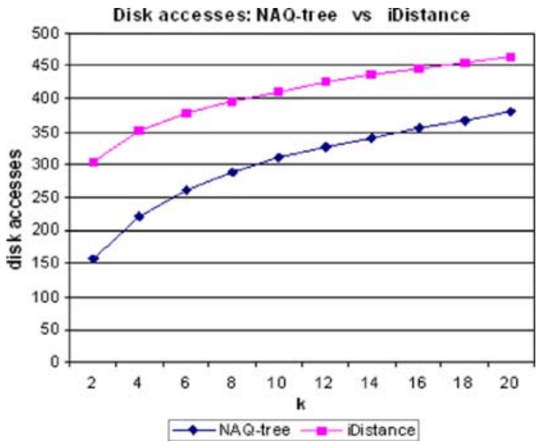


Fig. 20 Comparison of disk accesses using iDistance data set

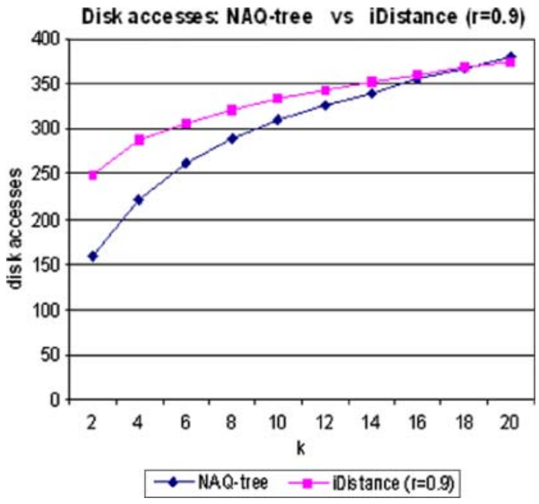


Fig. 21 Comparison of disk accesses after compaction using iDistance data set

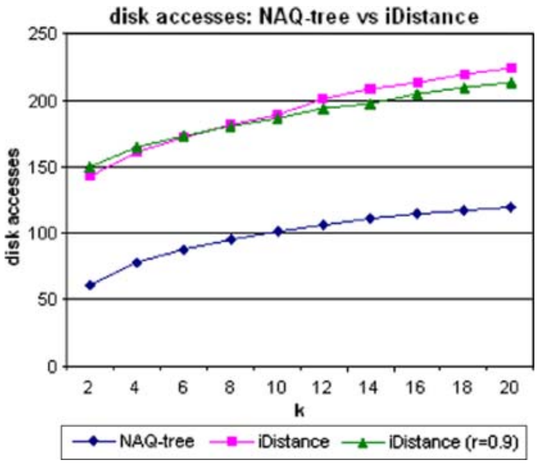


Fig. 22 Comparison of disk accesses after compaction using Phy_train data set

compacting to rate 90% for the iDistance data set. It shows that the compaction saved the disk accesses and made its performance closer to (still not as good as) NAQ-tree. However, the compaction does not always work so well; this is illustrated in Fig. 22, which compares the disk accesses of iDistance before and after compaction with NAQ-tree using Phy_train. For Phy_train, the number of disk accesses of iDistance after compaction is almost the same as that before compaction.

From Proposition 3.4, NAQ-tree can answer radius-limited similarity queries by one disk access if the radius is small. On the other hand, the number of disk accesses iDistance requires for such queries have been reported on Corel, Phy_train and idistance data set as 277, 99 and 241, respectively. In other words, if given a query, there exists in the database a perfectly matching point (or very close point), which is the most desirable one, NAQ-tree can get that point much faster than iDistance.

4.3 NAQ-tree versus DBM-tree

To compare the performance of NAQ-tree with DBM-tree as given in [29], we adjusted the parameters of NAQ-tree to be the same as those reported in [29] to produce the best performance of DBM-tree. The data set we used (Corel [10]) is the same as reported in [29].

Table 1 shows the number of visited nodes and visited leaf node of NAQ-tree for knn queries. Recall that an internal node is small and has fixed-size; it does not need to reside in its own page. The actual number of disk accesses can be less than the number of total visited nodes. The number of total visited nodes and the number of visited leaf nodes can be regarded as the upper and lower bounds of the disk accesses, respectively.

Figure 23 compares the disk accesses of the best case of DBM-tree with the disk accesses (upper and lower bounds) of NAQ-tree for knn queries. The curves plotted in Fig. 23 show that DBM-tree requires nearly 600 disk accesses for $k = 2$ and 780 disk accesses for $k = 20$, i.e., even in the worst case (that each node resides in one disk page), NAQ-tree can save 16–34% of the disk accesses.

For range queries, the work described in [29] tested ranges from 0.01 to 10% of the largest distance between pairs of data points in the data set. The number of disk accesses increases with ranges. The reported result showed that even for the smallest range (0.01%), the best DBM-tree requires over 260 disk accesses. This is caused by the inherent disadvantage of the

Table 1 NAQ-tree performance

k	Visited nodes	Visited leaf-nodes
2	387	190
4	470	253
6	513	287
8	542	310
10	566	328
12	586	345
14	605	359
16	622	373
18	637	385
20	651	396

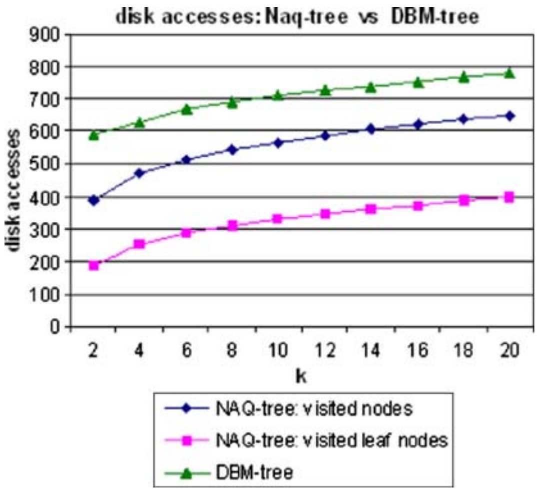


Fig. 23 Comparison of disk accesses

non-disjoint partitioning as discussed in Sect. 2. The query point may fall in the overlapped area of many partitions; all these partitions have to be searched no matter how small the query range is.

As the developers of DBM-tree did not give in [29] the value of the largest distance, we use the upper bound of the largest distance (in order to save time as it is very time-consuming to compute the actual largest distance for a data set of over 6×10^4 points); we tested ranges from 0.01 to 10% of the upper bound. (The method for computing the upper bound of the largest distance is presented in the appendix at the end of this paper). The results reported in Table 2 show that for the range from 0.01 to 4%, NAQ-tree visited from 43 to 256 nodes, i.e., the upper bound of disk accesses is less than 260. The best case of DBM-tree requires over 260 disk accesses even for the smallest range 0.01%. This highlights another advantage of NAQ-tree.

4.4 Scalability of NAQ-tree

To test the scalability of NAQ-tree as compared to VP-tree and iDistance, we decided to avoid using synthetic data in order to eliminate any possible bias in the testing. Consequently, we

Table 2 Range query performance of NAQ-tree

Range (%)	Visited nodes	No of points in answer set
0.01	43	0
0.10	44	0
0.50	51	1
1	63	1
1.50	78	2
2	100	4
2.50	128	6
3	165	8
3.50	203	10
4	256	13

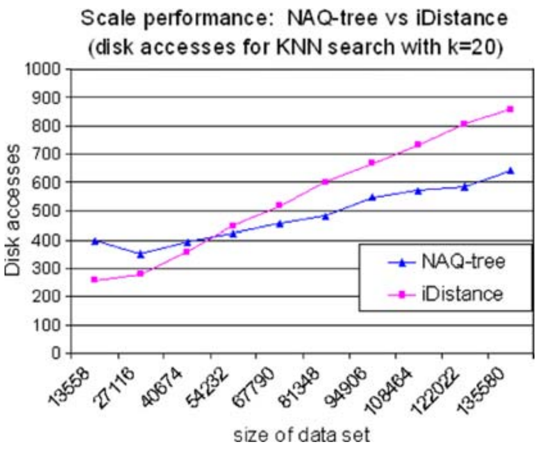


Fig. 24 Scalability of NAQ-tree versus iDistance

used real data, which we obtained by combining the two 32-dimensions data sets from [10] into one data set; then we divided the combined set into ten equal-sized parts. We started the experiment by using one of the ten parts as the data set and gradually added the remaining nine parts one by one until all the parts were included. We used the same set of queries to perform k-nearest neighbor search at each of the ten steps of the experiment. The achieved results are reported next in this section.

4.4.1 Scalability of NAQ-tree versus iDistance

To compare NAQ-tree with iDistance, we set the page capacity of NAQ-tree to be the same as that of iDistance; the developers of iDistance recommended using 64 reference points. We tested the number of disk accesses for k-nearest neighbor search by increasing the data set size from one part to ten parts. Figure 24 illustrates the number of disk accesses (for $k = 20$) required by NAQ-tree and iDistance. The curves plotted in Fig. 24 demonstrate that the number of disk accesses required by iDistance increases faster that NAQ-tree as the data set size increases. When the size of the data set is small (up to around 50K data points), iDistance performs a little better than NAQ-tree; however, as the data set size increases beyond 50K, the number of disk accesses required by NAQ-tree increases slower compared to iDistance, i.e.,

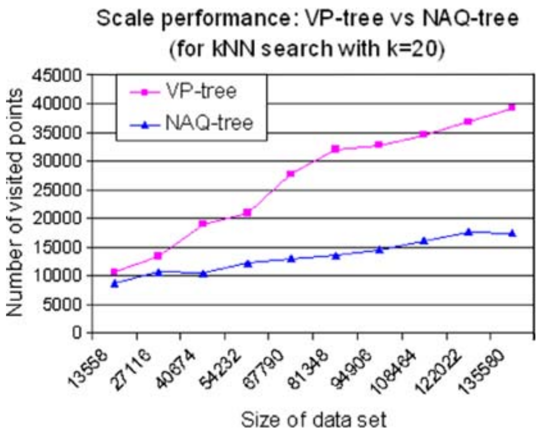


Fig. 25 Number of visited points: NAQ-tree versus VP-tree

NAQ-tree performs better than iDistance when the data set is larger than 50K. More importantly, the performance advantage of NAQ-tree over iDistance increases with the growth of data set. For example, using the data set with 135,580 points, NAQ-tree visits less than 650 pages, but iDistance requires visiting over 850 pages.

NAQ-tree performs better than iDistance because they differently handle the search radius r for k-nearest neighbor search. NAQ-tree starts with search radius $r = \infty$, and gradually decreases the value of r to the current kth nearest neighbor. On the other hand, iDistance starts from a small initial search radius r, and gradually increases r until k neighbors are located. In iDistance, the choice of the initial value of radius r and the increment at each step greatly affect the performance (i.e., setting the initial value of r to be close to the distance to the kth nearest neighbor will lead to good performance; however, the latter distance is unknown before the search). In iDistance, both parameters are determined based on the distance from query q to the nearest reference point. For fixed number of reference points, when the data set size increases, the distance between query q and the nearest reference point fails to give any insight for estimating appropriate values for both the initial radius r and the increment.

4.4.2 Scalability of NAQ-tree versus VP-tree

To compare NAQ-tree with VP-tree, we set the page capacity of NAQ-tree to be the same as that of VP-tree. We tested the number and percentage of visited points (for k-nearest neighbor search) as the data set size increases. Figures 25 and 26 compare NAQ-tree and VP-tree in terms of the number and percentage of visited points, respectively. Figure 25 shows that, when the data set size increases from 13,558 to 135,580, the number of points visited by NAQ-tree increased from around 9,000 to 17,000 (i.e., the increase is less than two times); whereas the number of points visited by VP-tree increased from around 10,000 to nearly 40,000 (*i.e.*, increased by almost four times). This demonstrates the performance advantage of NAQ-tree over VP-tree increases as the data set size increases. Figure 26 shows that both VP-tree and NAQ-tree visit decreasing percentage of points as the data set grows; the performance advantage of NAQ-tree over VP-tree is consistent.

NAQ-tree performs better than VP-tree because NAQ-tree deals better with the partitions. In the data partitioning, at each level, NAQ-tree maximizes the gaps between neighboring partitions; whereas in VP-tree, neighboring partitions are not separated. For a given search

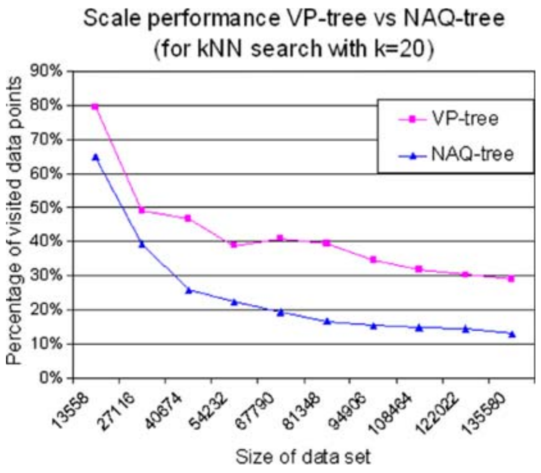


Fig. 26 Percentage of visited points: NAQ-tree versus VP-tree

radius r, NAQ-tree is more likely to search less partitions than VP-tree at each level. When the data set size increases, the height of the tree increases, thus the performance difference between NAQ-tree and VP-tree aggregates.

5 Conclusions and future work

In this paper, we proposed NAQ-tree as a new reference-based index structure for similarity search in high-dimensional metric space. NAQ-tree employs the idea of partitioning the data set into a set of nested approximate equivalence classes and confining the search space to small subset of partitions. NAQ-tree combines the advantages of the two types of different reference-based data partitioning methods (disjoint partition method like VP-tree, and non-disjoint cluster-based method like M-tree and DBM-tree). NAQ-tree partitions the data set in a disjoint way (so that no overlapping exists), while captures the inherent clustering property of the data distribution. The conducted tests demonstrate that NAQ-tree gives better performance than the major similarity search methods described in the literature.

For knn queries, all the index structures transform the query to range query with dynamic range. Given a query point q, if we know the distance r_k from q to its kth nearest neighbor, we may achieve the optimal performance by performing range query (q,r_k) to get the kth nearest neighbor. However, it is impossible to know r_k before the search is completed. In iDistance, search starts from a small range and iteratively increases until the range reaches r_k (*i.e.*, until k or more nearest neighbors are obtained). The initial search range and the increment value at each step are independent of the query, they are fixed for a given data set. However, for different queries, the distance to its kth nearest neighbor may vary greatly. Thus, it is impossible to find a fixed value of initial search range and fixed increments to provide good performance for any query. In the other hand, NAQ-tree starts the search from infinite range and the range is reduced to the distance of the current kth nearest neighbor at each step until the range reaches r_k. The search is guided by a heuristic that considers not only the data set, but the query as well. The current NAQ-tree can be regarded as using pessimistic heuristics that would not miss any nearest neighbors, but often over-estimates the search range such that it requires visiting more nodes than necessary. We are trying to further improve the performance of NAQ-tree by using more optimistic heuristics. In addition, the current NAQ-tree uses fixed search order of partitions at each level without considering the distance between partitions and the density of the partitions. We believe that it is possible to optimize the search order at each level and hence further improve the NAQ-tree performance. We are also working on extending the capabilities of the proposed approach to support for the deletion and insertion in the NAQ-tree. Although the current NAQ-tree is constructed using the full data set, it can easily be converted to a dynamic index by adding an insertion operation. As we do not need to keep the balance of the tree, in the insertion operation, one leaf node split will not propagate the change up to the root as it is the case with balanced trees, like the dynamic VP-tree and M-tree. Therefore, the insertion operation of NAQ-tree would be simpler and more efficient than insertion in balanced trees.

Appendix

Given a data set S with N d-dimensional points, and D is a distance metric defined on S

1. Compute the mean point M of the data set S; (time complexity is $O(dN)$)
2. Find the data point X that has largest distance to M; (time complexity is $O(dN)$)
3. Let $L = 2 \times D(X, M)$;

L is not less than the largest distance between the pairs of points in S. Formally, $\forall A, B \in S$, we have $D(A, B) \leq L$.

Proof

As X is the farthest point to M, we have:

$D(A, M) \leq D(X, M)$ and $D(B, M) \leq D(X, M)$

Thus, $D(A, B) \leq D(A, M) + D(B, M)$
(metric triangle inequality)
$\leq D(X, M) + D(X, M) = L$

Computing L is more efficient than computing the actual largest distance. In the experiments, we used L to calculate the query range (when comparing with DBM-tree).

References

1. An J, Chen H, Furuse K, Ohbo N (2005) CVA file: an index structure for high-dimensional datasets. Knowl Inform Syst 7:337–357
2. Andoni A, Indyk P (2008) Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. Comm ACM 51(1):117–122
3. Berchtold S, Keim D, Kriegel HP (1996) The X-tree: an index structure for high-dimensional data. In: Proceedings of VLDB
4. Beyer K et al (1999) When is “Nearest Neighbor” meaningful? In: Proceedings of IEEE ICDE
5. Bozkaya T, Ozsoyoglu M (1997) Distance-based indexing for high-dimensional metric spaces. In: Proceedings of ACM SIGMOD, pp 357–368
6. Brin S (1995) Near neighbor search in large metric spaces. In: Proceedings of VLDB, pp 574–584
7. Burkhard WA, Keller RM (1973) Some approaches to best-match file searching, Comm ACM 16(4):230–236
8. Chavez E, Navarro G (2005) A compact space decomposition for effective metric indexing. Pattern Recognit Lett 26(9): 1363–1376
9. Ciaccia P, Patella M, Zezula P (1997) M-Tree: an efficient access method for similarity search in metric spaces. VLDB J 426–435
10. Data set is available at: http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html
11. Cui B, Ooi BC, Su J, Tan T (2005) Indexing high-dimensional data for efficient in-memory similarity search. IEEE Trans Knowl Data Eng 17(3): 339–353

12. Dohnal V et al (2003) D-index: distance searching index for metric data sets. Multimedia Tools Appl 21(1)9:33
13. Filho RFS, Traina AJM, Traina C, Faloutsos C (2001) Similarity search without tears: the OMNI family of all-purpose access methods. In: Proceedings of IEEE ICDE, pp 623–630
14. Fu AW-C et al (2000) Dynamic VP-tree indexing for N-Nearest search given pair-wise distances. VLDB J
15. Source code is available at http://www.cs.mu.oz.au/~rui/code.htm
16. Jagadish HV, Ooi BC, Tan KL, Yu C, Zhang R (2005) iDistance: an adaptive B+-tree based indexing method for nearest neighbor search. ACM Trans Database Syst 30(2):364–397
17. Katamaya N, Satoh S (1997) The SR-tree: an index structure for high-dimensional nearest neighbor queries. In: Proceedings of ACM SIGMOD
18. Data set is available at: http://kodiak.cs.cornell.edu/kddcup/datasets.html
19. Kailing K, Kriegel H-P, Pfeifle M, Schonauer S (2006) Extending metric index structures for efficient range query processing. Knowl Inform Syst 10(2):211–227
20. Kadiyala S, Shiri N (2008) A compact multi-resolution index for variable length queries in time series databases. Knowl Inform Syst 15:131–147
21. Lowe DG (2004) Distinctive image features from scale invariant features. IJCV 60(2):91–110
22. Nievergelt J, Hinterberger H, Sevcik KC (1984) The grid file: an adaptable, symmetric multikey file structure. ACM Trans Database Syst 9(1)
23. Pawlak Z (1991) Rough sets theoretical aspects of reasoning about data. Kluwer, Dordrecht
24. Seidl T, Kriegel HP (1998) Optimal multi-step k-nearest neighbor search. In: Proceedings of ACM SIGMOD
25. Shaft U, Ramakrishnan R (2005) When is nearest neighbors indexable? In: Proceedings of ICDT, pp 158–172
26. Traina C, Traina AJM, Seeger B, Faloutsos C (2000) Slim-trees: highet performance metric trees minimizing overlap between nodes. In: Proceedings of EDBT, pp 51–65
27. Uhlmann JK (1991) Satisfying general proximity/similarity queries with metric trees. Inform Process Lett 40:175–179
28. Venkateswaran J, Lachwani D, Kahveci T, Jermaine C (2006) Reference-based indexing of sequences databases. In: Proceedings of VLDB
29. Vieira MR, Traina C, Chino FJT, Traina AJM (2004) DBM-tree: a dynamic metric access method sensitive to local density data. In: Proceedings of SBBD, pp 163–177
30. Source code is available at: http://www.cse.cuhk.edu.hk/~kdd/program.html
31. White DA, Jain R (1996) Similarity indexing with the ss-tree. In: Proceedings of IEEE ICDE
32. Weber R, Schek HJ, Blott S (1998) A quantitative analysis and performance study for similarity-search methods in high-dimensional space. In: Proceedings of VLDB
33. Yianilos P (1993) Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of ACM-SIAM symposium on discrete algorithms, pp 311–321
34. Yianilos P (1999) Excluded middle vantage point forests for nearest neighbor search. In: Implementation challenge, ALENEX'99

Author Biographies

Ming Zhang received the Master's degree in computer science from University of Regina, Canada, in 2005. He is currently a Ph.D. candidate in the Department of computer science at the Univerisity of Calgary. He published over ten papers in reputable international conferences and journals. His research interest includes database, data mining, XML and image processing. He is a recipient of NSERC Postgraduate Scholarship and iCore Postgraduate Scholarship.

Reda Alhajj received his B.Sc. degree in Computer Engineering in 1988 from Middle East Technical University, Ankara, Turkey. After he completed his B.Sc. with distinction from METU, he was offered a full scholarship to join the graduate program in Computer Engineering and Information Sciences at Bilkent University in Ankara, where he received his M.Sc. and Ph.D. degrees in 1990 and 1993, respectively. Currently, he is Professor in the Department of Computer Science at the University of Calgary, Alberta, Canada. He published over 275 papers in refereed international journals and conferences. He served on the program committee of several international conferences including IEEE ICDE, IEEE ICDM, IEEE IAT, SIAM DM; program chair of IEEE IRI 2008, OSIWM 2008, SONAM 2009, IEEE IRI 2009. He is editor in chief of *International Journal of Social Networks Analysis and Mining*, associate editor of IEEE SMC- Part C and he is member of the editorial board of the Journal of Information Assurance and Security; he has been guest editor for a number of special issues and edited a number of conference proceedings. Dr. Alhajj's primary work and research interests are in the areas of biocomputing and biodata analysis, data mining, multiagent systems, schema integration and re-engineering, social networks and XML. He currently leads a research group of seven Ph.D. and nine M.Sc. candidates. Dr. Alhajj recently received with Dr. Jon Rokne donation of equipment valued at $5 million from RBC and Teradata for their research on Computational Intelligence and Bioinformatics research.