Proc. 21st Int. Conf. on Scientific and Statistical Database Management (SSDBM'09), New Orleans, LA, 2009.

Reverse k-Nearest Neighbor Search based on Aggregate Point Access Methods

Hans-Peter Kriegel Peer Kröger
Matthias Renz Andreas Züfle Alexander Katzdobler

Institute for Computer Science
Ludwig-Maximilians-University of Munich
{kriegel,kroegerp,renz,zuefle,katzdobl}@dbs.ifi.lmu.de

Abstract. We propose an original solution for the general reverse k-nearest neighbor (RkNN) search problem in Euclidean spaces. Compared to the limitations of existing methods for the RkNN search, our approach works on top of Multi-Resolution Aggregate (MRA) versions of any index structures for multi-dimensional feature spaces where each non-leaf node is additionally associated with aggregate information like the sum of all leaf-entries indexed by that node. Our solution outperforms the state-of-the-art RkNN algorithms in terms of query execution times because it exploits advanced strategies for pruning index entries.

1 Introduction

For a given query object q, a reverse k-nearest neighbor (RkNN) query returns all objects of a database that have q among their actual k-nearest neighbors. In this paper, we focus on the traditional reverse k-nearest neighbor problem in feature databases and do not consider recent approaches for related or specialized RkNN tasks such as metric databases, the bichromatic case, mobile objects, etc. RkNN queries are important in many applications since the reverse k-nearest neighbors of a point p reflect the set of those points that are influenced by p. As a consequence, a considerable amount of new methods have been developed that usually extend existing index structures for RkNN search. The use of an index structure is mandatory in a database context because RkNN query processing algorithms are — like all similarity query processing algorithms — I/O-bound. The naíve solution for answering RkNN queries would compute for all objects of the database the k-nearest neighbors (kNN) and report those objects that have the query object on their kNN list. In order to present efficient solutions for RkNN search, most existing approaches make specific assumptions in order to design specialized index structures. Those assumptions include the necessity that the value of the query parameter k is fixed beforehand or the dimensionality of the feature space is low ($\leq$ 3). So far, the only existing approach for RkNN search that uses traditional (non-specialized) index structures and does not rely on the afore mentioned assumptions is called TPL [1]. In fact, the TPL approach computes a set of candidate points which is a superset of the result set in a first filter round. These candidates are used to prune other index entries already in this filter round. In a second refinement round, the kNNs of the candidates are computed to generate the final result.

In this paper, we extend the TPL approach in two important aspects. First, we generalize the pruning strategy implemented by the TPL approach by considering also other *entries* rather than only considering other *objects*. While the TPL approach usually needs to access several leaf nodes of the index although they may not include any true hits in order to start pruning other entries, we can start the pruning earlier and can save unnecessary refinements, i.e. disk accesses. Second, we show how entries may be pruned by themselves, which is a completely new pruning strategy not yet explored by the TPL approach. For this "self-pruning", we use the concept of aggregated point access methods like the aR-Tree [2, 3]. Furthermore, we show how both the enhanced and the new pruning strategies can be integrated into the original TPL algorithm by altering only a very limited number of steps. Because our novel RkNN search algorithm implements both the enhanced and the new pruning strategies, it is expected to prune more entries than the TPL approach, i.e. it produces less I/O overhead and reduces query execution times considerably.

The reminder of this paper is organized as follows. In Section 2 we formally define the RkNN problem and discuss recent approaches for solving this problem. Section 3 presents our novel RkNN query algorithm. Our new approach is experimentally evaluated and compared to the state-of-the-art approach using synthetic and real-world datasets in Section 4. Last but not least, Section 5 concludes the paper.

2 Survey

2.1 Problem Defintion

In the following, we assume that $\mathcal{D}$ is a database of n feature vectors, $k \leq n$, and $dist$ is the Euclidean distance[1] on the points in $\mathcal{D}$. In addition, we assume that the points are indexed by any traditional aggregate point access method like the aR-Tree family [2, 3].

The set of *k-nearest neighbors* of a point q is the smallest set $NN_k(q) \subseteq \mathcal{D}$ that contains at least k points from $\mathcal{D}$ such that

$$\forall o \in NN_k(q), \forall \hat{o} \in \mathcal{D} - NN_k(q) : dist(q, o) < dist(q, \hat{o}).$$

The point $p \in NN_k(q)$ with the highest distance to q is called the *k-nearest neighbor* (kNN) of q. The distance $dist(q, p)$ is called *k-nearest neighbor* distance (kNN distance) of q, denoted by $nndist_k(q)$.

The set of *reverse k-nearest neighbors* (RkNN) of a point q is then defined as

$$RNN_k(q) = \{p \in \mathcal{D} \mid q \in NN_k(p)\}.$$

The naive solution to compute the RkNN of a query point q is rather expensive. For each point $p \in \mathcal{D}$, the kNN of p is computed. If the distance between p and q is smaller or equal to the kNN distance of p, i.e. $dist(p, q) \leq nndist_k(q)$, then $q \in NN_k(p)$ which in turn means that point p is a RkNN of q, i.e. $p \in RNN_k(q)$. The runtime complexity of answering one RkNN query is $O(n^2)$ because for all n points, a kNN query needs to be launched which requires $O(n)$ when evaluated by a sequential scan. The costs of an RkNN query can be reduced to an average of $O(n \log n)$ if an index such as the R-Tree [4] or the R*-Tree [5]) is used to speed-up the kNN queries.

[1] Let us note that the concepts described here can also be extended to any L_p-norm.

2.2 Related Work

Here, we focus on feature vectors rather than on metric data. Thus, we do not consider approaches for metric data [6–9] as competitors. Usually, these approaches are less efficient on Euclidean data because they cannot make use of the Euclidean geometry. Existing approaches for the Euclidean RkNN search can be classified as self-pruning approaches or mutual-pruning approaches.

Self-pruning approaches are usually designed ontop of a hierarchically organized tree-like index structure. They try to estimate the kNN distance of each index entry E, i.e. E can be a database point or an intermediate index node. If the kNN distance of E is smaller than the distance of E to the query q, then E can be pruned. Thereby, self-pruning approaches do usually not consider other points (database points or index nodes) in order to estimate the kNN distance of an entry E but simply precompute kNN distances of database points and propagate these distances to higher level index nodes. The RNN-Tree [10] is an R-Tree-based index that precomputes for each point p the distance to its 1NN, i.e. $nndist_1(p)$ and index for each point p a sphere with radius $nndist_1(p)$ around p. The RdNN-Tree [11] extends the RNN-Tree by storing the points of the database itself in an R-Tree rather than circles around them. For each point p, the distance to p's 1NN, i.e. $nndist_1(p)$, is aggregated. For each intermediate entry E, the maximum of the 1NN distances of all child entries is aggregated. Since the kNN distances need to be materialized, both approaches are limited to a fixed value of k and cannot be generalized to answer RkNN-queries with arbitrary values of k. In addition, approaches based on precomputed distances can generally not be used when the database is updated frequently. Otherwise, for each insertion or deletion of points, the kNN distances of the points influenced by the updates need to be updated as well which is a considerably high computational overhead.

Mutual-pruning approaches use other points to prune a given index entry E. For that purpose, they use special geometric properties of the Euclidean space. In [12] a two-way filter approach for supporting R1NN queries is proposed that provides approximate solutions, i.e. may suffer from false alarms and incomplete results. A different approach is presented in [13] RkNN queries. Since it is based on a partition of the data space into equi-sized units where the border lines of the units are cut at the query point q and the number of such units increases exponentially with the data dimensionality, this approach is only applicable for 2D data sets. In [1] an approach for RkNN search was presented, that can handle arbitrary values of k and may be applied to arbitrary dimensional feature spaces. The method is called TPL and uses any hierarchical tree-based index structure such as an R-Tree to compute a nearest neighbor ranking of the query point q. The key idea is to iteratively construct Voronoi hyper-planes around q w.r.t. to the points from the ranking. Points and index entries that are beyond k Voronoi hyper-planes w.r.t. q can be pruned and need not to be considered for Voronoi construction. The idea of this pruning is illustrated in Figure 1 for $k = 1$. Entry E can be pruned, because it is beyond the Voronoi hyper-plane between q and candidate x. To decide whether an entry E can be pruned or not, TPL employs a special trimming function that examines if E is beyond k hyper-planes w.r.t. all current candidates. In addition, if E cannot be pruned but one or more hyper-planes intersect the page region of E, the trimming function trims the hyper-rectangular page region of E and, thus, potentially

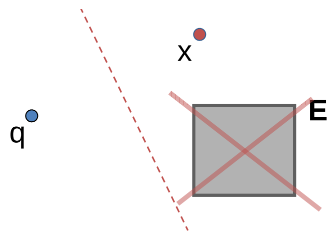


Fig. 1. TPL pruning ($k = 1$).

decreases the MinDist of E to q. As a consequence of such a trimming, E may move towards the end of the ranking queue when reinserted into this queue. This increases the chance that E can be pruned at a later step, because until then new candidates have been added. The remaining candidate points must be refined, i.e. for each of these candidates, a kNN query must be launched.

3 RkNN Search Using Multiple Pruning Strategies

3.1 Combining Multiple Pruning Strategies

As discussed above, we want to explore self-pruning as well as mutual pruning possibilities in order to boost RkNN query execution. Our approach is based on an index structure $\mathcal{I}$ for point data which is based on the concept of minimal-bounding-rectangles, e.g. the R-tree family including the R-tree [4], the R^*-tree [5] and the X-tree [14]. In particular, we use multi-resolution aggregate versions of these indexes as described in [2, 3] that e.g. aggregate for each index entry E the number of objects that are stored in the subtree with root E. The set of objects managed in the subtree of an index entry $E \in \mathcal{I}$ is denoted by *subtree*(E). Note that the entry E can be an intermediate node in $\mathcal{I}$ or a point, i.e. an object in $\mathcal{D}$. In the case that the entry $E \in \mathcal{I}$ is an object (i.e. $E = e \in \mathcal{D}$) then *subtree*$(E) = \{e\}$. The basic idea of our approach is to apply the pruning strategy mentioned above during the traversal of the index, i.e. to identify true drops as early as possible in order to reduce the I/O cost by saving unnecessary page accesses. The ability to prune candidates already at the directory level of the index implies that a directory entry is used to prune itself (self-pruning) or other entries (mutual-pruning).

For an RNN_k query with $k \geq 1$, an entry E can be pruned by another entry E' if there are at least k objects $e' \in$ *subtree*(E') such that E is behind the Voronoi hyperplane between q and e', denoted by $\perp(q, e')$. In general, we call a hyperplane $\perp(q, e)$ *associated with* the object e. Note, that a hyperplane $\perp(q, e)$ represents all points in the object space having equal distances to q and to e, i.e. for all points $p \in \perp(q, e)$ $dist(p, q) = dist(p, e)$ holds as shown in the example depicted in Figure 2. An object or point is called to be *behind* a hyperplane $\perp(q, e)$ if it is located within the half space determined by $\perp(q, e)$ which is opposite to the half space containing the query object q. Consequently, objects which are behind a hyperplane $\perp(q, e)$ are closer to e than to q, i.e. object o is behind $\perp(q, e)$ implies that $dist(o, q) > dist(o, e)$. Furthermore, an

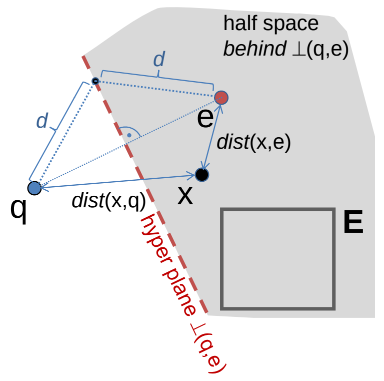


Fig. 2. Voronoi hyperplane between two objects q and e determining the half space which can be used to prune RkNN candidates.

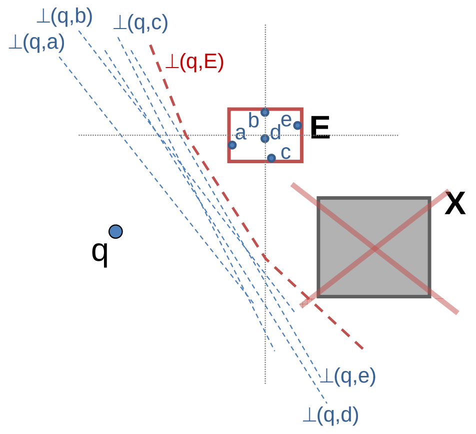


Fig. 3. Hyperplanes associated with all objects of an index entry E.

entry E (intermediate index node) is called to be *behind* a hyper plane $\perp(q, e)$, if all points of the entire page region of E are behind $\perp(q, e)$. In our example, object x as well as entry E are behind $\perp(q, e)$.

The key idea of the directory-level-wise pruning is to identify a hyperplane $\perp(q, E)$ which can be associated with an index entry E and which conservatively approximates the hyperplanes associated with all objects e in the subtree of E, i.e. $e \in \mathit{subtree}(E)$. Figure 3 illustrates the idea of this concept. We say that the hyperplane associated with an index entry E is *related to* the set of objects in the subtree of E. Since we assume that the number of objects stored in the subtree of an index entry E is known, we also know for the hyperplane associated with that index entry E, $\perp(q, E)$, how many objects this hyperplane relates to. We can use this information in order to prune entries according to E without accessing the child entries of E. For example, if the number of objects that relate to $\perp(q, E)$ is greater than the query parameter k, we can prune

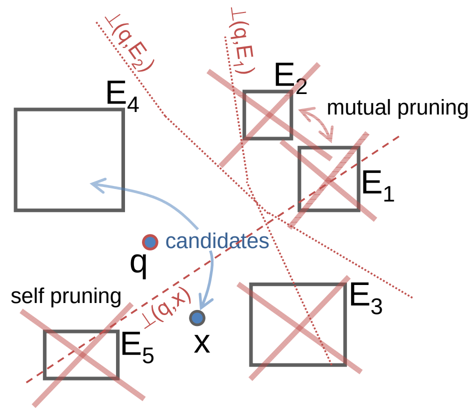


Fig. 4. Pruning potentials using different pruning strategies.

all points and entries that are behind $\perp(q, E)$. In Figure 3, entry X can be pruned for $k \leq 5$ because $|\mathit{subtree}(E)| = 5$. As mentioned above, to obtain the number of objects stored in a subtree of an entry, we can exploit the indexing concept as proposed in [2]. This concept allows to store for each index entry E the number of objects stored in the subtree that has E as its root, i.e. the aggregate value $|\mathit{subtree}(E)|$ is stored along with each entry E. For example, the aggregate R-tree (aR-tree) [2, 3] is an instance of this indexing concept. Then, the number of objects that are related to $\perp(q, E)$ equals $|\mathit{subtree}(E)|$.

In addition, we can use these considerations also for the self-pruning of entries. If an entry stores more than k objects in its subtree, i.e. $|\mathit{subtree}(E)| > k$, and E is behind the hyperplane that is associated with itself, $\perp(q, E)$, then E can be pruned. The rational for this is that $|\mathit{subtree}(E)| > k$ objects relate to $\perp(q, E)$, i.e. more than k hyperplanes are approximated by $\perp(q, E)$. As a consequence, each object $o \in \mathit{subtree}(E)$ is behind at least k hyperplanes. This self-pruning can be performed without considering any other entry. For example, for $k \leq 4$, the entry E in Figure 3 can also be pruned without considering any other entry because each point in the subtree of E is behind the hyperplane of all four other points in E.

Figure 4 visualizes the benefits of using higher level mutual-pruning and self-pruning on a fictive 2D Euclidean database indexed by an R-Tree-like structure. The hyperplane associated with an index entry E is denoted by $\perp(q, E)$. If we assume that each of the entries E_1, E_2, and E_5 stores more than k objects in its particular subtree, all three entries can be pruned by the self-pruning strategy which does not consider any other entries. This can be done because all three entries are lying behind those hyperplanes that are associated with themselves. On the other hand, entries E_1, E_2 and E_3 can be pruned by the mutual-pruning strategy which is based on heuristics that consider other entries. While E_3 can only be pruned for $k = 1$ due to the hyperplane associated with object x, both E_1 and E_2 even can be pruned for $k \geq 1$ with the assumption that each of the values of $|\mathit{subtree}(E_1)|$ and $|\mathit{subtree}(E_2)|$ is greater than or equal to k. Let us note that a mutual-pruning approach like [1] needs at least one exact object to prune other entries, i.e. E_1, E_2 and E_5 can neither prune themselves nor prune each other. In that case, only entry E_3 could be pruned and all other entries need to be refined. The extension of the mutual-pruning strategy and the combination with the self-pruning strategy allows us to prune all candidates except for E_4 and object x. This simple example illustrates the potential benefit of our approach. In other words, the aim of our novel method is to provide the advantages of the mutual-pruning and the self-pruning approaches by fading out the drawbacks of both, thus, providing the "best of two worlds". As a consequence our solution is expected to outperform the existing approach [1] in terms of query execution times because of the advanced pruning capabilities that are derived from the combination of the self-pruning and mutual-pruning potentials on higher index levels.

3.2 Intermediate Index Entry Hyperplanes

The most important question is, how to derive a hyperplane $\perp(q, E)$ associated with an entry $E \in \mathcal{I}$. This hyperplane $\perp(q, E)$ associated with an index entry E is required to constitute a conservative approximation of the hyperplanes associated with all objects o in the subtree of E, i.e. $o \in$ *subtree*(E). In fact, we will see that $\perp(q, E)$ is defined by means of a set of hyperplanes rather than by only one hyperplane, depending on the location in the feature space. In general, a set of hyperplanes H is called *conservative approximation* of another set of hyperplanes H', if all objects related to the hyperplane $h \in H$ are definitely behind each hyperplane $h' \in H'$, formally:

$$(\forall h \in H : o \text{ behind } h) \Rightarrow (\forall h' \in H' : o \text{ behind } h')$$

An example is shown in Figure 5, where the hyperplane $\perp(q, E)$ associated with the index entry E forms a conservative approximation of all hyperplanes that are associated with the objects covered by E. The hyperplane approximation consists of the three hyperplanes h_1, h_2 and h_3. Objects that are behind these three hyperplanes, e.g. object o, are definitely behind all hyperplanes that are associated with the objects covered by E, independent of their location in E. Such an approximation is sensible if we assume that the set H is much smaller than the set H' and, thus, can be used to prune entries more efficiently.

In the following, we show how we can define such a set of hyperplanes $\perp(q, E)$ associated with an index entry E which conservatively approximates the hyperplanes of all objects stored in the subtree of E. As mentioned above, a hyperplane associated with an object o represents all points p which have the same distance to the query point q and to o. In addition, we know that all objects stored in the subtree of an index entry E are located inside the minimum bounding hyper-rectangle (mbr) that defines the page region of E. Thus, we can determine a conservative hyperplane representation of all points stored in the subtree of entry E if we replace the distances between the hyperplane points $p \in \perp(q, E)$ and $o \in$ *subtree*(E) by the maximum distance between p and the mbr-region of E. Figure 6 illustrates the computation of such a conservative approximation for a given index entry E in a two-dimensional feature space. First, we have to specify the maximum distance between an mbr-region of the index entry E and any point in the vector space. It suffices to find for each point p in the vector space the point e within the mbr-region which has the maximum distance to p. This can be done

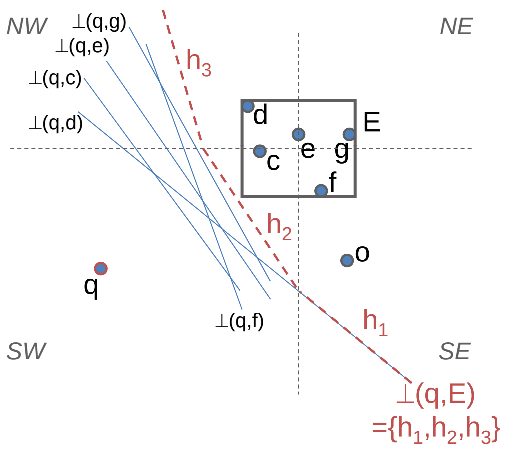


Fig. 5. Conservative approximation $\perp(q, E)$ of the hyperplanes associated with all objects of an index entry E.

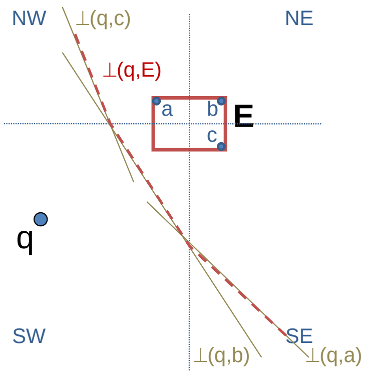


Fig. 6. Computation of conservative hyperplane approximations.

by considering partitions of the vector space which are constructed as follows: in each dimension the space is split paraxially at the center of the mbr-region. As illustrated for the two-dimensional example in Figure 6, we obtain partitions denoted by NW, NE, SE and SW. In each of these partitions P, the corner point of the mbr-region which lies within the diagonally opposite partition is the mbr-region point which has the maximum distance to all points within P. In our example, for any point p in SW the maximum distance of p to E is the distance between p and point b in partition NE. Consequently, the hyperplane $\perp(q, b)$ is a conservative approximation of all hyperplanes between points within the mbr-region of E and the points within the partition SW. This way, in our example the hyperplane associated with E is composed of the three hyperplanes $\perp(q, a)$, $\perp(q, b)$ and $\perp(q, c)$. Generally, the conservative approximation of an mbr-region in a d-dimensional space consists of at most 2^d hyperplanes. This is due to the

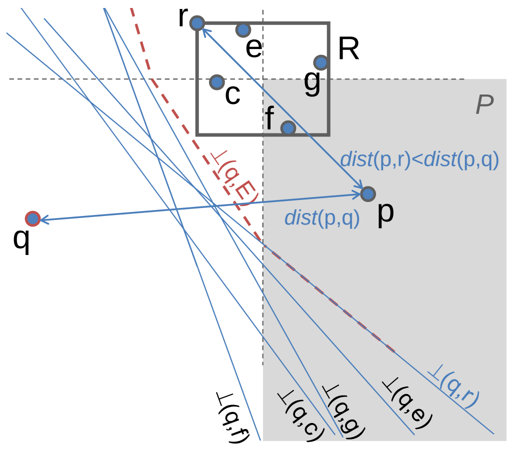


Fig. 7. Conservative approximation $\perp(q, E)$ of the hyperplanes associated with all objects of an index entry E.

fact that a d-dimensional space can be partitioned into 2^d partitions according to an mbr-region, such that the maximums distance between each point p in a partition and an mbr-region E is defined by exactly one point within E.

In the following we will show that if we construct the hyperplane approximation as mentioned above we achieve in fact a conservative hyperplane approximation. The example illustrated in Figure 7 visualizes the scenario described in the following lemma.

Lemma 1. *Let q be a (query) point, R be an mbr-region in a d-dimensional space and p be any d-dimensional point. Furthermore, let P denote the space partition which is generated by splitting the space paraxially at the center of the mbr-region R in each dimension and let P be the partition containing the point p. Then, the hyperplane between q and the corner point r of R which lies within the diagonally opposite partition builds a conservative approximation of all hyperplanes between q and all other points in R within the partition P. In other words, all points in P that are behind $\perp(q, r)$ are also behind each hyperplane between q and any other point in R.*

Proof. By definition, each point p behind the hyperplane $\perp(q, r)$ has a smaller distance to the point r than to q, i.e. $dist(p, r) < dist(p, q)$. Furthermore, r is assumed to be the point in R with the maximal distance to p, i.e. the distance from p to any point p' in R is smaller or equal to $dist(p, r)$. Consequently, the distance between p and p' is smaller than the distance between p and q. Since the hyperplane $\perp(q, p')$ associated with any point p' in R only contains points having equal distance to p' and q by definition, $\perp(q, p')$ cannot contain such a point p which is assumed to be behind $\perp(q, r)$. As a consequence, no hyperplane associated with q and any point in R is behind $\perp(q, r)$ within the region P.

According to Lemma 1, we can combine all hyperplane approximations of all regions associated with an mbr-region R into a set of hyperplanes that conservatively approximate the hyperplanes of all points in R w.r.t. the entire data space. The two-dimensional example illustrated in Figure 5 shows that the combination of the three hyperplanes, marked by the red dotted poly-line, conservatively approximates the hyperplanes associated with all points within E. Each index entry X which lies behind the hyperplane approximation $\perp(q, E)$ associated with the entry E also lies behind each of the hyperplanes associated with each object in E.

Note, that the shape of the hyperplane associated with an mbr-region depends on the topology between the query point q and the mbr-region. Figure 8 exemplarily shows four cases with different topologies in the two-dimensional space. Case 1 (cf. Figure 8(a)) shows the standard case where the hyperplane associated with E can be represented by three regular hyperplanes, i.e. three hyperplanes each associated with a single point. As shown in case 2 (cf. Figure 8(b)), the hyperplane associated with E is generally represented by four regular hyperplanes. Case 3 (cf. Figure 8(c)) shows the special scenario where q is located at one of the corner points of the mbr-region. Here, the hyperplane associated with E is represented by only two regular hyperplanes. An interesting case is case 4 where none of the four regular hyperplanes of which the hyperplane associated with E is constructed are applicable. The reason is that none of the four regular hyperplanes which are used to construct the hyperplane associated with E intersects the region it relates to and, thus, cannot be used to prune any candidate.

3.3 Pruning Candidates

In the following, we show how the hyperplanes $\perp(q, E)$ associated with an index entry E can be used to prune itself or other entries. Here, we assume that an RkNN-query with $k \geq 1$ is issued. Since, the hyperplanes $\perp(q, E)$ associated with an index entry E approximate all hyperplanes associated with all objects within the subtree of E, any index entry X which lies behind $\perp(q, E)$ in fact must lie at least behind $|subtree(E)|$ hyperplanes, and, thus can be pruned by E if $|subtree(E)| \geq k$. For this reason, we assign a weight $w(\perp(q, E)) \in \mathbb{N}^+$ to each hyperplane $\perp(q, E)$ associated with an index entry E. The weight $w(\perp(q, E))$ denotes the number of hyperplanes which are approximated by $\perp(q, E)$. Since, we use aggregate index structures, we assume that the number of objects managed by an index entry E is accessible without the need to refine the entry E. Once, the hyperplanes $\perp(q, E)$ for an entry E are built, we can assign the weight $w(\perp(q, E)) = \mathit{subtree}(E)$ to them. Obviously, a hyperplane associated with an object has the weight 1.

In summary, if we assume that we already determined a set of n hyperplanes $\mathcal{S} = \{\perp(q, E_1), \perp(q, E_1), \cdots, \perp(q, E_n)\}$ behind which an index entry E lies, then, we can prune E, if $n \geq k$.

3.4 The RkNN Search Algorithm

The benefit of our formalization presented above is that the two pruning strategies, general mutual-pruning and self-pruning, can now easily be integrated into the TPL algorithm. In other words, our concepts allow us to use the TPL algorithm as a framework for RkNN search. Thus, our novel algorithm also relies on a filter step and a refinement

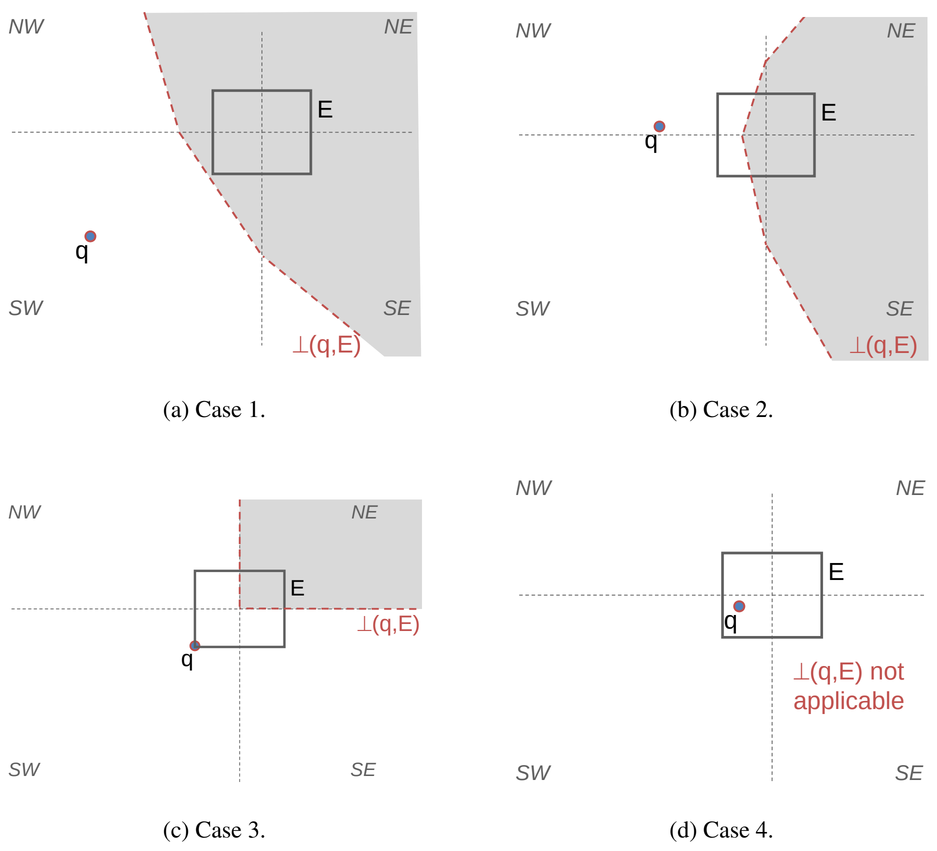


Fig. 8. Conservative hyperplane approximations associated with point q and mbr-region E for different cases w.r.t. the topology of q and E.

step. Our filter step is very similar to the filter step of the TPL approach. We also manage a heap H to compute a nearest neighbor ranking, a set of candidates points S_{cnd} and a set of pruned entries S_{rfn}. The key difference is that we call the trimming function in a different way. Instead of trimming an index entry or a database point w.r.t. the candidate points in S_{cnd}, we use all entries/points in H, S_{rfn}, S_{cnd} for trimming. This implements the advanced mutual-pruning already on the directory level of the index as well as the self-pruning of index entries. In addition, we have to generalize the trimming function such that the clipping of page regions considers the weight of each hyperplane. The clipping algorithm sketched in [1] can easily be adapted for this purpose. Finally, our refinement step is algorithmically also very similar to the refinement step of TPL. However, it is expected that it requires less disc accesses because usually less candidates need to be refined. Intuitively, the refinement step tests for each point in S_{cnd} if

Name	size	dimension	distribution
Synth1	3,500	2	uniform
Synth2	3,500	2	6 Gaussians
Synth3	1,000,000	2	6 Gaussians
Synth4	1,500	20	uniform

Table 1. Features of the synthetic data sets used for evaluation.

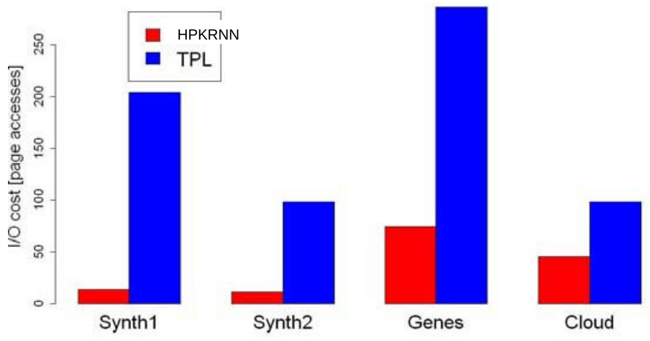


Fig. 9. Comparison of HPKRNN and TPL processing R1NN queries.

the query q is among its kNN list by considering and iteratively refining the points and index entries in S_{rfn}.

4 Experimental Evaluation

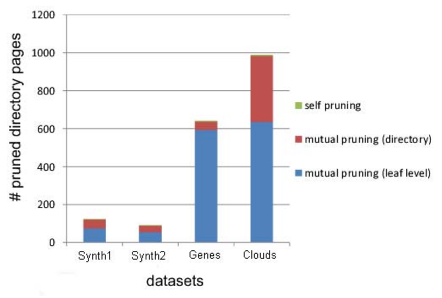
We compared our novel approach for RkNN search, hereafter referred to as HPKRNN (short for Hyper-Plane based RkNN), with TPL [1] the current state-of-the-art algorithm. All experiments are based on an R*-Tree with a page sizes of 32 Byte and 1KByte. For all experiments, we executed 100 sample RkNN queries and averaged the results.

Our experiments are conducted on four synthetic data sets with different features that are summarized in Table 1.

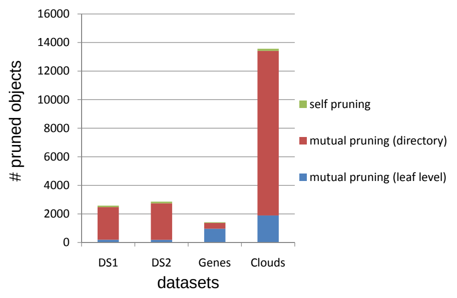
Additionally, we conducted experiments on two real-word data sets. The “Genes” data set contains appr. 5,000 points in a 5D space representing the expression levels of genes. The data set “Cloud” contains 9D weather parameters recorded at appr. 17,100 different locations in Germany.

4.1 Evaluation of the I/O-Cost

Figure 9 displays the performance of the competitors on four data sets when processing R1NN queries using an R*-tree with a page size of 32 byte. We used Synt1 and Synt2 because they feature different characteristics, as well as the two real-world data sets Genes and Cloud. It can be seen that our novel HPKRNN algorithm significantly



(a) Pruning power w.r.t. directory nodes.



(b) Pruning power w.r.t. data objects.

Fig. 10. Benefit of different pruning strategies for HPKRNN.

outperforms the TPL approach in terms of I/O costs and, thus, query execution times. The reason for this clear performance boost over the mutual-pruning approach TPL can be derived from Figure 10 where the number of self-pruned objects, the number of mutual-pruned objects on the leaf level, and the number of mutual-pruned objects on higher levels in the index are displayed separately for our HPKRNN approach. As it can be observed from this figure, the combination of the pruning strategies on multiple levels as performed by HPKRNN is beneficial and superior over using only mutual-pruning on the leaf level of the index as it is done by TPL. Especially when considering pruned objects, the big positive effect of the mutual-pruning at the directory level becomes obvious (cf. Figure 10(b)). But also for pruning directory pages, especially the mutual-pruning at the directory level erases a large number of candidates (cf. Figure 10(a)). It can also be observed from both charts in Figure 10, that contribution of the self-pruning strategy seems to be less important on the applied data sets.

Next, we evaluated the scalability of the competitors w.r.t. the number of data objects n. Figure 11 displays the results for an R*-Tree with a pagesize of 32Byte and Figure 12 reports the results for an R*-Tree with a pagesize of 1kB. Again, the performance gain of our HPKRNN algorithm over the TPL method remains significant with varying number of data objects. In particular for large databases our method outperforms the TPL method by up to two orders of magnitude.

In the next experiments, we evaluated the impact of the query parameter k on the scalability of the competitors. The resulting performances are visualized in Figure 13. Again, our method clearly outperforms the TPL approach on most data sets especially for smaller values of k. With increasing k, the gap between both approaches decreases. A reason for this might be that the self-pruning and the mutual-pruning at the directory level becomes less selective in this case. Rather, with increasing k, most directory pages and objects are pruned with the mutual-pruning at the leaf level. However, as observable from the Synt1 and Gene data sets, this effect is only visible for rather high values of k.

Figure 14 shows the influence of the query parameter k when using an R*-Tree with a larger page-size (in this case 1K). Here, we evaluated the number of page accesses (cf.

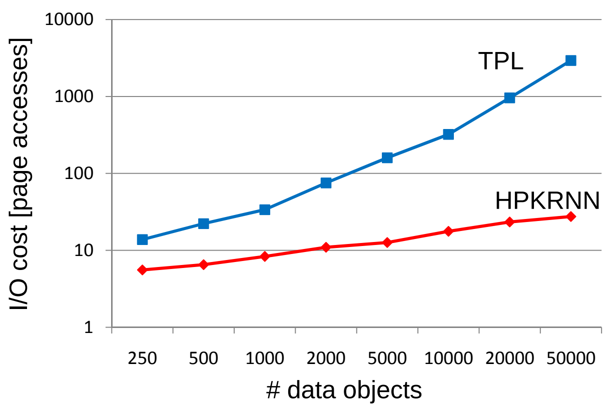


Fig. 11. Scalability of the competitors w.r.t. the data set size using an R*-tree with a page size of 32 Byte.

Figure 14(a)) and the maximum size of the candidate set (cf. 14(b)) produced by each method. The relatively large number of page accesses of the TPL algorithm (Figure 14(a)) can be explained by a rapidly growing number of entries in the candidate set of the TPL algorithm, shown in Figure 14(b). Additionally, it can be observed in Figure 14(a) that the vast majority of page accesses in both approaches occur in the respective filter steps whereas the refinement round requires only a small number of page accesses. This is an interesting observation because HPKRNN is clearly superior to TPL in the filter step (cf. Figure 14(a)) and additionally in the number of objects that need to be refined (cf. Figure 14(b)), i.e. produces a considerably smaller number of candidates with a significantly smaller amount of page accesses.

We also evaluated the scalability of our approach w.r.t. the number of dimensions d of the data set using the 20-dimensional data set “Synth4” containing 1,500 data points. Figure 15 shows the results of the experiment in which we subsequently increased the number of relevant dimensions. In this experiment, we chose a fixed capacity of data points that can be stored in an R*-Tree node to keep the results comparable, in particular, a capacity of 30 data points for directory nodes and a capacity of 60 data points for data (leaf) nodes.

It can be observed that our HPKRNN algorithm outperforms the TPL method for dimensions less or equal to five, i.e. $d \leq 5$. For higher dimensions, both approaches appear to perform very similar. This can be explained by the general bad performance of R*-Trees on more than 5-dimensional data. In order for an minimal bounding rectangle (mbr) to contain its minimal number of entries, it has to cover an increasingly large fraction of space in each dimension.

4.2 Evaluation of the CPU-Cost

Next, we evaluated the time required to compute the results of RkNN-queries with respect to the database size in terms of CPU-time. We also compared the CPU costs of our

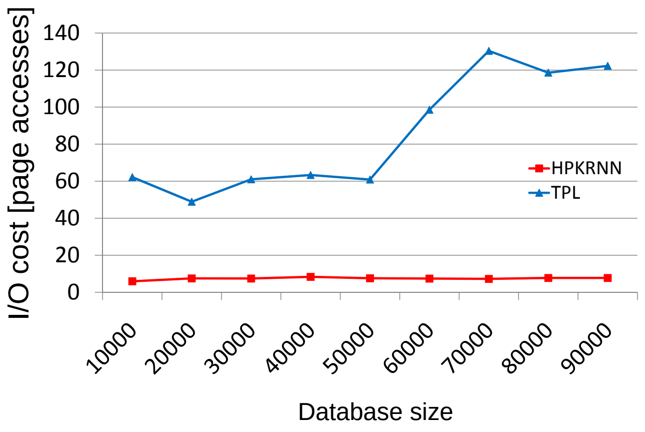
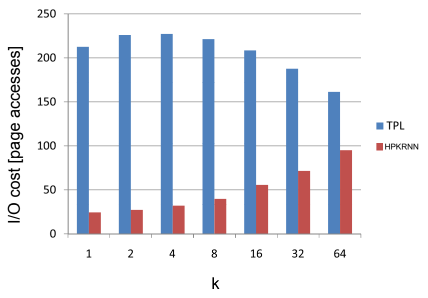


Fig. 12. Scalability of the competitors w.r.t. the data set size using an R*-tree with a page size of 1024.

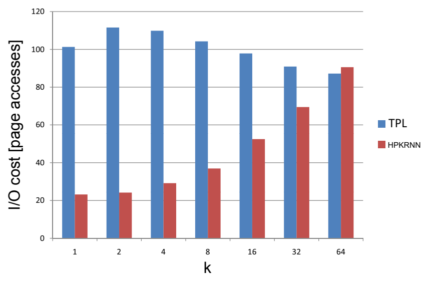
HPKRNN approach to the CPU costs of the TPL approach. Here, we only compared the time required for the refinement step, because, in [1], the CPU costs of the filter step is boosted using heuristics based on Hilbert values. Since our TPL algorithm does not implement this heuristic, we decided to omit experiments on the runtime of the filter step of the TPL, in order to avoid unfair comparisons. Note that not using these heuristics proposed in the TPL approach does not affect the I/O costs of the TPL approach. The result of this experiment is shown in Figure 16 for different values of k. It can be seen that our method is competetive with the TPL approach in terms of CPU runtime. This indicates that, since we need more effort to compute hyper-planes between the query and a directory node, a less number of hyper-planes is needed to prune objects and nodes. This coincides directly with the observation made above that our HPKRNN method produces less candidates in the filter step because the number of hyper-planes computed is determined by the number of candidates we have during the filter step.

4.3 Summary

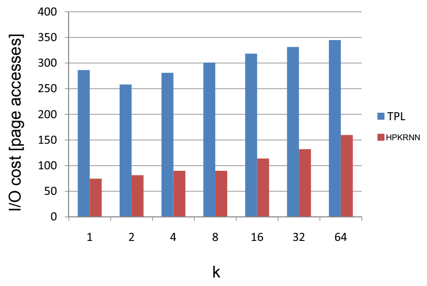
In summary, the conducted experiments confirm that our novel approach clearly outperforms the current state-of-the-art approach because it combines multiple pruning strategies rather than implementing only one pruning paradigm. Even though our novel pruning strategies may produce an additional overhead at the CPU end, we also showed that the CPU costs of our algorithm is competitive with the CPU costs of the existing method. Furthermore, our approach saves a considerably number of page accesses during the filter step and the refinement step. As a consequence, since the RkNN problem is I/O-bound for large data sets, our new algorithm needs significantly less time to report the results of RkNN queries than the current state-of-the-art approach for this problem.



(a) Synth1.



(b) Synth2.

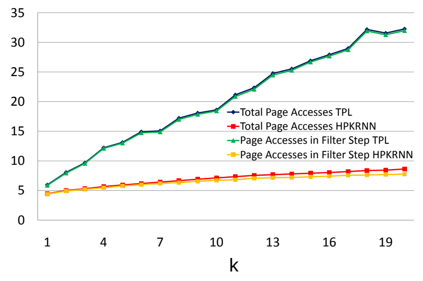


(c) Gene.

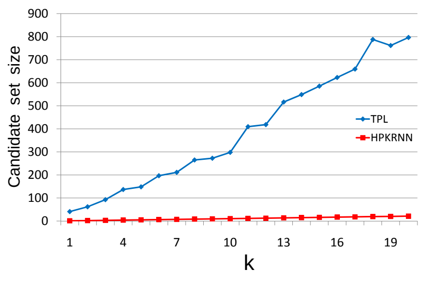
Fig. 13. Performance of the competitors in the number of pages accessed w.r.t. different values of k using an R*-Tree with a pagesize of 32 Byte.

5 Conclusions

In this paper, we propose a generalization of the TPL algorithm which is the current state-of-the-art approach to Euclidean RkNN search. Our solution extends the TPL method in two important ways. First, the mutual-pruning strategy of TPL is generalized so that it can be applied already on higher levels of the index. Second, we introduced a new pruning paradigm called self-pruning. The generalization of the mutual-pruning strategy and its combination with the new self-pruning strategy helps to explore the full pruning potentials in order to reduce query execution times. Our experimental evaluation confirms that our new solution outperforms the existing methods significantly in terms of query execution times.



(a) Number of page accesses.



(b) Size of the candidate set.

Fig. 14. Performance of the competitors w.r.t. different values of k using an R*-Tree with a page-size of 1024 Byte.

References

1. Tao, Y., Papadias, D., Lian, X.: Reverse kNN search in arbitrary dimensionality. In: Proc. VLDB. (2004)
2. Lazaridis, I., Mehrotra, S.: Progressive approximate aggregate queries with a multi-resolution tree structure. In: Proc. SIGMOD. (2001)
3. Papadias, D., Kalnis, P., Zhang, J., Tao, Y.: Efficient olap operations in spatial data warehouses. In: Proc. SSTD. (2001)
4. Guttman, A.: R-Trees: A dynamic index structure for spatial searching. In: Proc. SIGMOD. (1984) 47–57
5. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-Tree: An efficient and robust access method for points and rectangles. In: Proc. SIGMOD. (1990) 322–331
6. Achtert, E., Böhm, C., Kröger, P., Kunath, P., Pryakhin, A., Renz, M.: Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In: Proc. SIGMOD. (2006)
7. Achtert, E., Böhm, C., Kröger, P., Kunath, P., Pryakhin, A., Renz, M.: Approximate reverse k-nearest neighbor search in general metric spaces. In: Proc. CIKM. (2006)
8. Achtert, E., Böhm, C., Kröger, P., Kunath, P., Pryakhin, A., Renz, M.: Efficient reverse k-nearest neighbor estimation. In: Proc. BTW. (2007)
9. Tao, Y., Yiu, M.L., Mamoulis, N.: Reverse nearest neighbor search in metric spaces. IEEE TKDE **18**(9) (2006) 1239–1252
10. Korn, F., Muthukrishnan, S.: Influenced sets based on reverse nearest neighbor queries. In: Proc. SIGMOD. (2000)
11. Yang, C., Lin, K.I.: An index structure for efficient reverse nearest neighbor queries. In: Proc. ICDE. (2001)
12. Singh, A., Ferhatosmanoglu, H., Tosun, A.S.: High dimensional reverse nearest neighbor queries. In: Proc. CIKM. (2003)
13. Stanoi, I., Agrawal, D., Abbadi, A.E.: Reverse nearest neighbor queries for dynamic databases. In: Proc. DMKD. (2000)
14. Berchtold, S., Keim, D.A., Kriegel, H.P.: The X-Tree: An index structure for high-dimensional data. In: Proc. VLDB. (1996)

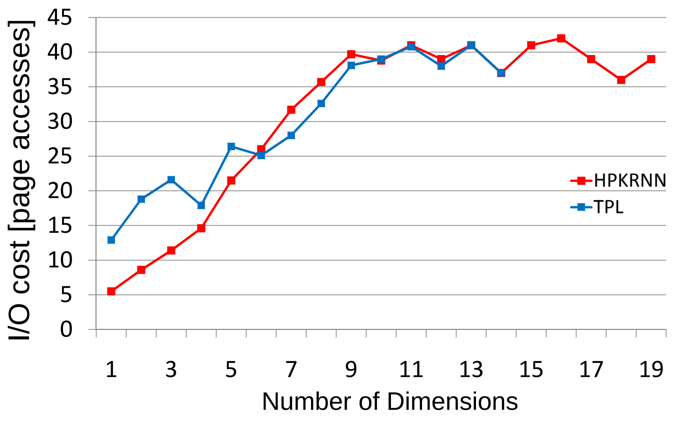


Fig. 15. Performance of the competitors w.r.t. the number of dimensions d of the data set.

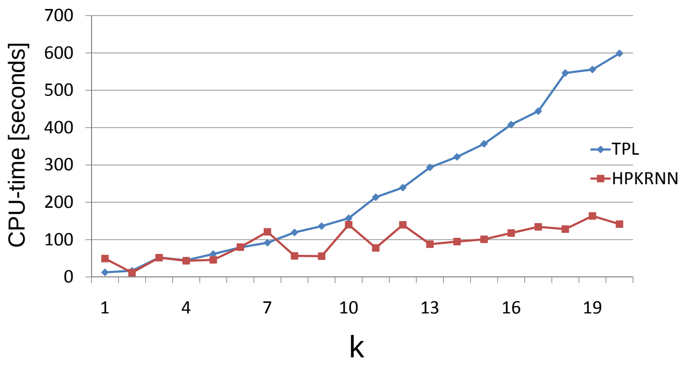


Fig. 16. Scalability of the competitors in terms of CPU costs w.r.t. k.