Reverse k-Nearest Neighbor Search in Dynamic and General Metric Databases

Elke Achtert   Hans-Peter Kriegel   Peer Kröger   Matthias Renz   Andreas Züfle

Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München, Germany
`http://www.dbs.ifi.lmu.de`
{achtert,kriegel,kroegerp,renz,zuefle}@dbs.ifi.lmu.de

ABSTRACT

In this paper, we propose an original solution for the general reverse k-nearest neighbor (RkNN) search problem. Compared to the limitations of existing methods for the RkNN search, our approach works on top of any hierarchically organized tree-like index structure and, thus, is applicable to any type of data as long as a metric distance function is defined on the data objects. We will exemplarily show how our approach works on top of the most prevalent index structures for Euclidean and metric data, the R-Tree and the M-Tree, respectively. Our solution is applicable for arbitrary values of k and can also be applied in dynamic environments where updates of the database frequently occur. Although being the most general solution for the RkNN problem, our solution outperforms existing methods in terms of query execution times because it exploits different strategies for pruning false drops and identifying true hits as soon as possible.

1. INTRODUCTION

A reverse k-nearest neighbor (RkNN) query returns for a given query object q 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 and do not consider recent approaches for related or specialized reverse nearest neighbor tasks such as the bichromatic case, mobile objects, etc.

Since the efficient support of traditional RkNN queries is important in many applications involving Euclidean and general metric data [14], this topic has received growing attention recently. As a consequence, a considerable amount of new methods have been developed that usually extend existing index structures for RkNN search. However, all these methods for RkNN search suffer at least from one of the following drawbacks. First, some approaches are only applicable to a fixed value of k (typically $k = 1$) or at most to a specific range $1 \leq k \leq k_{max}$ of values that needs to be specified in advance. Second, many approaches are only applica-

ble for static scenarios where no objects are inserted into or deleted from the database because the costs for an update of the underlying specialized index structure are extremely high. Third, some approaches are tailored to Euclidean or metric spaces only because they use specific geometric properties of the Euclidean space or specific properties of metric indexes.

In addition, although all approaches more or less use index structures, none of them explore the full potentials of pruning index entries already on the directory level. This is a key limitation because RkNN query processing algorithms are — like all similarity query processing algorithms — I/O-bound. Thus, applying indexes is mandatory but each directory entry of the index that needs to be refined implies an I/O-intensive disc access. In fact any (directory or leaf) entry E of an index can be pruned if the query objects q cannot be one of the k-nearest neighbors. Existing approaches explore this pruning property of an index entry in two differently ways. First, one part of approaches try to estimate the k-nearest neighbor distance of each entry E considering precomputed distances of the objects contained in the sub-tree represented by E or other heuristics that are tailored to specific properties of E like the number of objects contained in the sub-tree representing E or the extension of the page region represented by E, etc. However, none of these approaches consider other entries in order to estimate the k-nearest neighbor distance of an entry. Second, the other part of approaches try to identify candidate results in order to prune index entries that have a smaller distance to k candidates than to q. In other words, they consider other objects in order to prune entries. However, they cannot prune an entry because of another directory entry (i.e. they always need other fully refined *database objects* for pruning) and an entry E can also not be pruned without considering other entries/objects e.g. by using an estimation of its k-nearest neighbor distance.

Obviously, both approaches complement each other but each approach for its own may require more index node refinements invoking page accesses and, thus, may produce higher I/O costs than necessary because it does not use all potentials for pruning.

In this paper, we propose to combine both approaches to achieve optimal pruning power and, thus, reduce query execution time. In addition, through jointly using both pruning paradigms we are able to combine their advantages by fading out their further drawbacks not directly related to their pruning strategy. As a consequence, to the best of

our knowledge, this paper is the first contribution to solve the generalized RkNN search problem for arbitrary metric objects in a dynamic environment which is not tailored to a specific data type or index structure. In particular, our method provides the following new features:

1. It can be applied to general metric objects, i.e. databases containing any type of complex objects as long as a metric distance function is defined on these objects. It does not use any specific index structure but works with any hierarchically organized, tree-like access method.

2. It is applicable to the generalized RkNN problem where the value of k is specified at query time. Thereby, the maximal applicable value of k is not constrained by any threshold k_{max}.

3. It does not rely on the precomputation of distances and, thus, has no additional costs for inserting or deleting objects from the database. Updates are processed by the underlying index structure.

4. Due to the combination of both pruning-paradigms, our approach — although being the most general solution — even outperforms the existing approaches designed for a more specialized problem in terms of query execution times.

The reminder of this paper is organized as follows. In Section 2 we formally define the RkNN problem we want to solve here, discuss recent approaches for this problem, and point out our contributions. Section 3 explores how both existing pruning strategies can be combined and extended to achieve more pruning power during index traversal. In Section 4 these basic ideas are implemented to answer RkNN queries efficiently. Our novel approach is experimentally evaluated and compared to existing approaches using synthetic and real-world datasets in Section 5. Last but not least, Section 6 concludes the paper.

2. SURVEY

2.1 Problem Defintion

Since we focus on the traditional reverse k-nearest neighbor problem, we do not consider recent approaches for related or specialized reverse nearest neighbor tasks such as the bichromatic case, mobile objects, etc. In the following, we assume that $\mathcal{D}$ is a database of n objects, $k \leq n$, and $dist$ is a metric distance function on the objects in $\mathcal{D}$. The set of k-nearest neighbors of an object q is the smallest set $NN_k(q) \subseteq \mathcal{D}$ that contains at least k objects 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 object $p \in NN_k(q)$ 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 an object 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 object q is rather expensive. For each object $p \in \mathcal{D}$, 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 object 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 objects, 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 M-Tree [5] (or, if the objects are feature vectors, the R-Tree [6] or the R*-Tree [4]) is used to speed-up the kNN queries.

2.2 Related Work

Existing approaches for the RkNN search can be classified as self-pruning approaches or mutual-pruning approaches.

Self-pruning approaches are usually designed on top of a hierarchically organized tree-like index structure. They usually try to estimate the kNN distance of each index entry E, i.e. E can be a database object 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 usually estimate the kNN distance of each index entry E by only considering special properties of E rather than taking also other objects (database objects or index nodes) into account. Several approaches use an exact estimation by simply precomputing kNN distances. The RNN-Tree [7] is an R-Tree-based index that precomputes for each object p the distance to its 1NN, i.e. $nndist_1(p)$. The objects are not stored in the index itself. Rather, for each object p, the RNN-Tree manages a sphere with radius $nndist_1(p)$, i.e. the data nodes of the tree contain spheres around objects. The RdNN-Tree [15] extends the RNN-Tree by storing the objects of the database itself in an R-Tree rather than circles around them. For each object p, the distance to p's 1NN, i.e. $nndist_1(p)$, is aggregated. For each leaf entry E, the maximum of the 1NN distances of all objects in E is aggregated. An inner node of the RdNN-Tree again aggregates the maximum 1NN distances of all its child nodes. Thus, both the RNN-Tree and the RdNN-Tree allow for pruning false hits already on the directory levels of the underlying R-Tree. Due to the materialization of the 1NN distances of all data objects, both approaches need not to compute 1NN queries for the remaining objects that are not pruned because those objects contribute to the final result. In addition, the RdNN-Tree, can be extended to metric spaces (e.g. by applying an M-Tree instead of an R-Tree). However, 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. To overcome this problem, the MRkNNCoP-Tree [2] has been proposed which is conceptually similar to the RdNN-Tree but stores a conservative and progressive approximation for all kNN distances of any data object rather than the exact kNN distance for one fixed k. The only limitation is that k is constrained by a parameter k_{max} specifying the maximal value of k that can be supported. For RkNN queries with $k > k_{max}$, the MRkNNCoP-Tree cannot be applied. The conservative and progressive approximations of any index node are propagated to the parent nodes. Using these approximations, the MRkNNCoP-Tree can identify a candidate set, true hits, and true drops. For each object in the candidate set, a kNN query is launched for refinement. A variant of the MRkNNCoP-Tree is proposed in [1, 3] that achieves a further runtime improvement and gets rid of the k_{max} bound for the value of k on the one hand but on the other hand generates only approximative results, i.e. the re-

sults may contain false hits and are not guaranteed to be complete. A different approach is proposed in [14] where a method for RkNN search in metric spaces that is tailored to the M-Tree is presented. The authors derive several rules from the M-Tree structure that can be used to estimate the kNN distance of an index entry. Each remaining database object o that is contained in a non-pruned leaf node of the M-Tree needs to be refined, i.e. a kNN query needs to be computed for o.

Mutual-pruning approaches are usually designed for the Euclidean space only and use other objects to prune a given index entry e. For that purpose, they use special geometric properties of the Euclidean space, typically the concept of Voronoi cells. The basic idea is that given the Voronoi-cell around the query object q, each object or index node E can be pruned if E is beyond a Voronoi plane (for $k = 1$). In [11] a two-way filter approach for supporting R1NN queries based on this idea is proposed. The method provides approximate solutions, i.e. may suffer from false alarms and incomplete results. In [13] the first approach for RkNN search was presented, that can handle arbitrary values of k. The method uses any hierarchical tree-based index structure such as an R-Tree to compute a nearest neighbor ranking of the query object q. The key idea is to iteratively construct a Voronoi cell around q from the ranking. Objects that are beyond k Voronoi planes w.r.t. q can be pruned and need not to be considered for Voronoi construction. The remaining objects must be refined, i.e. for each of these candidates, a kNN query must be launched. In [12], a different approach for R1NN search in a 2D data set is presented. It is based on a partition of the data space into six equi-sized units where the border lines of the units are cut at the query object q. The 1NN of q in each unit is determined and all these neighbors are merged together to generate a candidate set. This considerably reduces the cost for the nearest-neighbor queries. The candidates are then refined by computing for each candidate c the nearest neighbor. Since the number of units in which the candidates are generated increases exponentially with the data dimensionality, this approach is only applicable for 2D data sets.

2.3 Discussion and Contributions

All recent methods for RkNN search suffer at least from one of the following drawbacks. Self-pruning approaches that rely on precomputed kNN distances are only applicable for a fixed value of k (typically $k = 1$) or at least for a specific range $1 \leq k \leq k_{max}$ of values that needs to be specified before index construction. In addition, the costs for reconstructing the indexes in case of an update of the database are very high. Thus, approaches that use precomputed distances disqualify for a general solution. Self-pruning approaches that rely on the estimation of the kNN distances of objects or index entries from other heuristics than precomputed distances have less pruning power and do not use the distances to other index entries in order to get more accurate estimations. As a consequence, subtrees of an index need to be refined although they might have been pruned because the estimation of their kNN distances is too less accurate (but would have been accurate enough when considering other entries). On the other hand, mutual-pruning approaches use geometric properties of the Euclidean space like Voronoi cells that do not exist in general metric spaces and, thus,
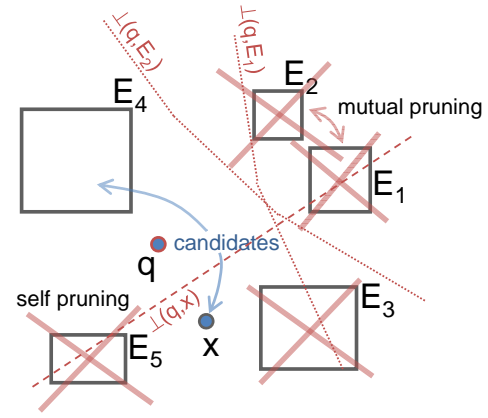

Figure 1: Illustration of pruning potentials using a fictive Euclidean dataset indexed by an R-Tree (see text for details).

cannot be extended for metric databases. In addition, they do not make use of the potentials of self-pruning, i.e. they may need to traverse a subtree of the index although it cannot qualify for the query due to an estimation of its kNN distance (which may be derived without any precomputing). In addition, both pruning strategies cannot identify true hits but need additional refinement rounds. Even though these refinement rounds can usually be processed rather efficiently, each approach gives away the potentials of the other pruning approach.

In this paper, we try to combine the potentials of self-pruning (without any precomputation and materialization of distances) and mutual-pruning in order to reduce the page accesses during index traversal and producing the results of a given RkNN query in one run through the index. Figure 1 visualizes the benefits of such an approach on a fictive 2D Euclidean database indexed by an R-Tree-like structure. When using a self-pruning strategy without precomputation that estimates the kNN distance of each entry from heuristics that do not consider other entries (e.g. from the extend of the region of the corresponding entry), we will be able to prune only entries E_5 and E_2. Entry E_1 has to be refined although it could be pruned when considering E_2. On the other hand, a mutual-pruning approach like [13] needs at least one exact object to prune other entries, i.e. E_2 and E_1 cannot prune each other. Here, only E_3 could be pruned. Entry E_5 will never be pruned by any object in other entries, so E_5 needs to be refined anyway. However, in fact all entries except E_4 could be pruned in this situation when combining both pruning strategies and extending the mutual-pruning also to intermediate entries. 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 — to the best of our knowledge — this paper contributes the first solution for the generalized RkNN search problem applicable for arbitrary metric data, not tailored to a specific index structure. Since our approach does not require any RkNN search specific access methods and no distance pre-computation it is qualified also in a dynamic database environment where updates may oc-
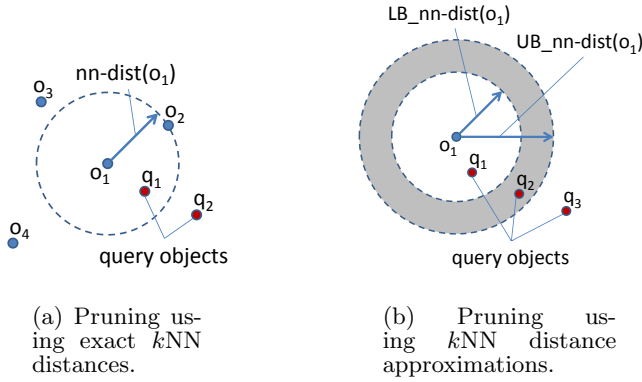
(a) Pruning using exact kNN distances.

(b) Pruning using kNN distance approximations.

Figure 2: Illustration of self-pruning.

(a) Pruning using objects.

(b) Pruning using index entries (here: R-Tree entries).

Figure 3: Illustration of mutual-pruning.

cur frequently. In addition, our very general solution even outperforms the existing approaches that are designed for a more specialized problem 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.

3. RKNN SEARCH USING MULTIPLE PRUNING STRATEGIES

3.1 Combining Multiple Pruning Strategies

As discussed above, we want to explore both self-pruning as well as mutual pruning possibilities. Our approach is based on an arbitrarily hierarchically organized tree-like index structure $\mathcal{I}$, e.g. a R*-tree [4] or an M-tree [5]. 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 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 and true hits as early as possible in order to reduce the I/O cost by saving unnecessary page accesses.

The existing self-pruning works as follows: Each object $o \in \mathcal{D}$ can be pruned if its kNN distance is smaller than the distance of o to the query object q. Otherwise o is a true hit. Figure 2(a) illustrates this strategy. Object o_1 is one of the RkNNs of query object q_1 because $nndist_k(o_1) \geq dist(o_1, q_1)$ while it is not one of the RkNNs of query object q_2 because $nndist_k(o_1) \geq dist(o_1, q_2)$. However, as discussed above, precomputation of all kNN distances for all possible values of k is too expensive or even impossible and would additionally result in heavy computational costs for each update of the database. Rather, we have to try to approximate the kNN distances of each database object using heuristics. Two kinds of heuristics are helpful. First, an upper bounding kNN distance approximation $UB\_nn\text{-}dist(o)$ for which $nndist_k(o) \leq UB\_nn\text{-}dist(o)$ holds can be used to prune true drops because if the upper bounding kNN distance approximation is smaller than the distance of o to the query object q, i.e. $dist(q, o) > UB\_nn\text{-}dist(o)$, then o can be pruned. Second, a lower bounding kNN distance approximation $LB\_nn\text{-}dist(o)$ for which $LB\_nn\text{-}dist(o) \leq nn\text{-}dist(o)$
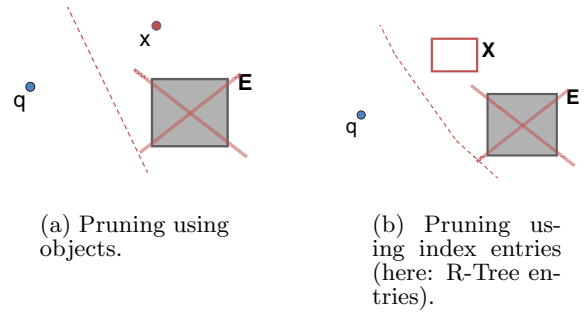
holds can be used to identify true hits because if the lower bounding kNN distance approximation is greater or equal than the distance of o to the query object q, i.e. $dist(q, o) \leq LB\_nn\text{-}dist(o)$, then o is one of qs RkNNs. If $LB\_nn\text{-}dist(o) < dist(q, o) < UB\_nn\text{-}dist(o)$ we cannot make a decision on o. In that case we have to try to get better approximations or even compute the exact kNN distance of o. An example for these decision rules is shown in Figure 2(b). While object o_1 is an RkNN of q_1 but not an RkNN of q_3, it cannot be decided whether o is an RkNN of q_2 or not. In fact, without considering other objects or entries, approximations of the kNN distances are usually hard to obtain for objects. On the other hand, for intermediate entries in the underlying index structure, we can derive upper and lower bounds based on the properties of the page region, e.g. [14] derives such approximations for M-Tree entries. In this paper, we will introduce general methods to derive distance approximations for an entry E that are not tailored to any index structure and additionally considers information about the distance of E to other entries in the index. The later can be rather beneficial because this way, a more accurate approximation of the kNN distance of an entry can be computed.

The mutual-pruning strategy basically works with a similar idea — approximating the kNN distance of an entry, or, more precisely, estimating the kNNs of an entry. The major difference to the self-pruning approach is that it considers the location of neighboring objects to estimate the kNNs of an entry. So far, the existing implementations can only be applied to Euclidean data: given the Voronoi cell around q which is specified by a set of objects $V_q \subseteq \mathcal{D}$, we know that all objects $o \notin V_q$ and corresponding entries E that are outside of this cell can be pruned when searching for the R1NN of q because at least one of the objects in V_q that builds the Voronoi cell has a smaller distance to o or all objects in E than q Figure 3(a) illustrates this idea: an object x builds a Voronoi hyperplane (dashed line) and entry E can be pruned because it is beyond that line w.r.t. q. This idea can be extended to the general RkNN case by pruning objects that are beyond k Voronoi hyperplanes. In order to apply this strategy also for metric objects, we first generalize this idea to metric spaces. Generally, given a Voronoi hyperplane H_x between q and an object $x \in \mathcal{D}$ in a Euclidean space, the fact that object $o \in \mathcal{D}$ "is beyond H_x" can be expressed by $dist(x, o) < dist(q, o)$. This expression no longer uses the Euclidean geometry but only relys on distances and, thus, can be applied to general metric objects. As a consequence, for a R1NN query we can prune each (general metric) object

$o \in \mathcal{D}$ and each entry E of a metric index structure if there is another object $x \in \mathcal{D}$ such that $dist(x,o) < dist(q,o)$ and $dist(x,e) < dist(q,e)$ for all objects $e \in subtree(E)$. Generalizing this to RkNN search, we need k of such objects $x \in \mathcal{D}$ in order to prune o or E as true drop. A second extension to the original mutual-pruning approach is the following. So far, object x must be a database object. Thus, before the pruning can start, at least one object of the database (usually the NN of q) needs to be obtained from the index, i.e. at least one path of the index root to the leaf needs to be completely refined to get a candidate object x even though x need not necessarily be an RkNN of q. This is a limitation because, even on higher levels of the index we may generally be able to perform mutual pruning. We generally may find for a given entry $E \in \mathcal{I}$ k other *entries* $X \in \mathcal{I}$ such that for all objects $o \in subtree(E)$ there is an object $x \in subtree(X)$ with $dist(x,o) < dist(q,o)$. This situation can be formalized by the following inequality:

$$LB\_dist(E,q) > UB\_nn\text{-}dist(E),$$

where $LB\_dist(E,q)$ is a lower bound of the distance of all objects $o \in subtree(E)$ to q and $UB\_nn\text{-}dist(E)$ is an upper bounding estimation of the kNN distances of all objects $o \in subtree(E)$.

Let us note that this rule is equivalent to the pruning rule of the self-pruning strategy using the lower bounding approximations for the kNN distance and the distance to the query but already on a higher level of the index. If E is an object $(subtree(E) = e)$, then obviously $LB\_dist(e,q) = dist(e,q)$ and $UB\_nn\text{-}dist(e) = nndist_k(e)$. Analogously, we can define a rule to identify true hits using the lower bounding bounding kNN distance approximation of all objects in $subtree(E)$, denoted by $LB\_nn\text{-}dist(E)$, and an upper bound of the distance between E and q, denoted by $UB\_dist(E,q)$.

Combining both pruning strategies, we end up with the following rules: Assuming that the data objects are indexed by any hierarchical index structure $\mathcal{I}$, an entry $E \in \mathcal{I}$

1. is a true hit if $UB\_dist(E,q) < LB\_nn\text{-}dist(E)$,

2. is a true drop if $LB\_dist(E,q) > UB\_nn\text{-}dist(E)$.

In summary, we have contributed the following advances over existing approaches so far. We have first combined the two separately existing pruning strategies into two rather general pruning rules for any entry E that rely on abstract upper and lower bounding approximations of (i) the distance between E and q and (ii) the kNN distance of E. Especially the estimations of the latter one are not constrained, i.e. can be obtained from considering the structure of the page region of E as well as the distance of E to other entries. Second, we have extended the mutual-pruning concept such that it can be applied to general metric data and to the pruning at higher levels during the index traversal.

3.2 Computing Distance Approximations

The derived pruning rules rely on abstract distance approximations. In the following, we derive suitable distance functions to estimate such distance approximations as accurate as possible considering properties of the page regions and the distance to other entries simultaneously.

Let q be a query object and $E, E' \in \mathcal{I}$ be entries representing either directory nodes or objects. Our pruning strategies are based on the lower bounding and upper bounding esti-

mation of the following distances: we assume that we are able to estimate

- the distances of all objects in $subtree(E)$ to the query object q in order to derive an estimation for $LB\_dist(E,q)$ and $UB\_dist(E,q)$;

- the distances of all objects in $subtree(E)$ to all objects in $subtree(E')$ in order to derive an estimation for $LB\_nn\text{-}dist(E)$ and $UB\_nn\text{-}dist(E)$ (as well as $LB\_nn\text{-}dist(E')$ and $UB\_nn\text{-}dist(E')$, respectively);

- the distances between all objects in $subtree(E)$ in order to derive an estimation for $LB\_nn\text{-}dist(E)$ and $UB\_nn\text{-}dist(E)$.

In the following, we define distance approximations for these distances.

DEFINITION 1 (DISTANCE APPROXIMATIONS). *The following distance functions are defined on two index entries E and E' with the following properties:*

$MinDist(E,E')$ *always underestimates the distances between the objects in $subtree(E)$ and $subtree(E')$:*
$\forall o \in subtree(E), \forall o' \in subtree(E') :$
$$MinDist(e,e') \leq dist(o,o').$$

$MaxDist(E,E')$ *always overestimates the distances between the objects in $subtree(E)$ and $subtree(E')$:*
$\forall o \in subtree(E), \forall o' \in subtree(E') :$
$$MaxDist(E,E') \geq dist(o,o').$$

$MinMaxDist(E,E')$ *is the minimal overestimation of the distances between the objects in $subtree(E)$ and $subtree(E')$:*
$\forall o \in subtree(E), \exists o' \in subtree(E') :$
$$MinMaxDist(E,E') \geq dist(o,o').$$

$MaxMinDist(E,E')$ *is the maximal underestimation of the distances between the objects in $subtree(E)$ and $subtree(E')$:*
$\forall o \in subtree(E), \exists o' \in subtree(E') :$
$$MaxMinDist(E,E') \leq dist(o,o').$$

The definition of these distance approximations also works if E and E' are identical, i.e. $subtree(E) = subtree(E')$. In that case, these distance functions estimate the kNN distances of $E = E'$ without considering other entries. In addition, if $E' = q$ the distance functions approximate the distance of E to q.

In the following we present implementations of these four distance approximations exemplarily for the case that the regions of intermediate index entries are rectangular like in R-Tree variants and for the case that the regions of intermediate index entries are based on a covering radius and a routing object like in M-Tree variants. Distance approximations based on M-tree nodes and R-tree nodes are illustrated in Figure 4.

3.2.1 Implementations for R-Tree Nodes

Since, the minimum and maximum distance approximations between objects and node regions as well as between two node regions are rather intuitive and already defined in several publications concerning indexing, e.g. [10], here we omit them due to space limitations. Rather, we want to concentrate on the more specialized distance functions $MinMaxDist$ and $MaxMinDist$.
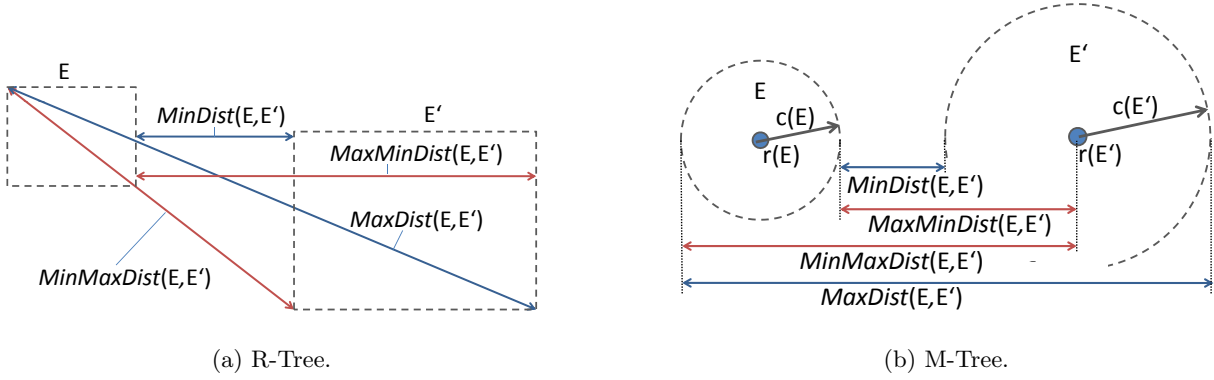
Figure 4: Illustration of the distance approximations for Euclidean and metric index structures.

A definition of the *MinMaxDist* distance function defined for an object and an R-Tree region can also be found in [10]. The *MinMaxDist* function defined for two intermediate R-tree entries E and E' is quite similar to the *MinMaxDist* definition given in [10]. We assume that the page region of an entry E which is a d-dimensional hyper-rectangle is specified by its lower left corner $(E_1.l, \ldots, E_d.l)$ and upper right corner $(E_1.r, \ldots, E_d.r)$. Furthermore, the center of the page region is denoted by the vector $(E_1.m, \ldots, E_d.m)$ with $E_i.m = (E_i.l + E_i.r)/2$. The *MinMaxDist* function defined for E and E' can be computed as follows[1]:

$$ MinMaxDist(E, E') = $$

$$ \sqrt[p]{\min_{1 \le i \le d} ((pE_i - pE_i')^p + \sum_{j=1, j \ne i}^{d} d_{max}(E_j, E_j'))}, $$

where $d_{max}(E_j, E_j') = \max\{(E_j.u - E_j'.l)^p, (E_j.l - E_j'.u)^p\}$,

$$ pE_i' = \begin{cases} E_i'.l & \text{, if } E_i.m < E_i'.m \\ E_i'.u & \text{, if } E_i.m \ge E_i'.m \end{cases} $$

and

$$ pE_i = \begin{cases} E_i.l & \text{, if } E_i.m < pE_i' \\ E_i.u & \text{, if } E_i.m \ge pE_i' \end{cases} $$

Note that here we assume that the distance function *dist* is any L_p-norm. The intuition behind the above formula is to find for each dimension $i \in \{1, \ldots, d\}$ one of the two border hyper-planes of the rectangle associated with E' which are orthogonal to the dimension i and is closer to the farthest point $p \in subtree(E)$, i.e. p is enclosed by the rectangular region of E. Finally, the function *MinMaxDist* computes the maximum distance between p and the corresponding hyper-plane (cf. Figure 4(a)).

The function *MaxMinDist* can be defined analogously. Here, we define *MaxMinDist* for two index entries $E \in \mathcal{I}$ and $E' \in \mathcal{I}$ regardless whether an entry E is an object or an intermediate index node. This can be done, since an object can be considered as a specialized region with no extension.

$$ MaxMinDist(E, E') = $$

[1]The nodes E and E' are assumed to be dissimilar, i.e. $subtree(E) \ne subtree(E')$.

$$ \sqrt[p]{\max_{1 \le i \le d} ((pE_i - pE_i')^p + \sum_{J=1, j \ne i}^{d} d_{min}(E_j, E_j'))}, $$

where $d_{min}(E_j, E_j') = \max\{0, (E_j.l - E_j'.u)^p, (E_j'.l - E_j.u)^p\}$,

$$ pE_i' = \begin{cases} E_i'.u & \text{, if } E_i.m < E_i'.m \\ E_i'.l & \text{, if } E_i.m \ge E_i'.m \end{cases} $$

and

$$ pE_i = \begin{cases} E_i.l & \text{, if } E_i.l > pE_i' \\ E_i.u & \text{, if } E_i.u < pE_i' \\ pE_i' & \text{, else} \end{cases} $$

Figure 4(a) also illustrates all distance approximations for the sample R-Tree entries. The special case where both entries $E \in \mathcal{I}$ and $E' \in \mathcal{I}$ are identical, i.e. $subtree(E) = subtree(E')$ have to be considered separately. For an entry $E \in \mathcal{I}$, we can only estimate $MinMaxDist(E, E) = MaxDist(E, E)$ and $MaxMinDist(E, E) = MinDist(E, E)$. For example, for the $MinMaxDist(E, E)$ estimation, in the worst case we have to assume an object $o \in subtree(E)$ is located at one corner of E and all other objects in $subtree(E)$ are located at the opposite corner of E. The distance between two opposite corners of an entry E obviously equals $MaxDist(E, E)$. The minimum distance can be derived analogously. Let us note that both the *MinMaxDist* and the *MaxMinDist* functions — unlike *MinDist* and *MaxDist* — are not symmetric in general.

3.2.2 Implementations for M-tree Nodes

For the distance estimations defined on M-tree entries we do not need to distinguish between intermediate M-tree entries and objects. The region of an entry E is defined by means of a rooting object $r(E) \in subtree(E)$ and a covering radius $c(E)$ such that for all objects $o \in subtree(E)$ it holds that $dist(o, r(E)) \le c(E)$. Each object $o \in \mathcal{D}$ defines a degenerated region with routing object o and the covering radius $c(o) = 0$. Again, we omit the specification of the well-known *MinDist* and *MaxDist* distance approximations for M-tree entries due to space limitation. The *MinMaxDist* function defined for two M-tree entries $E \in \mathcal{I}$ and $E' \in \mathcal{I}$ is defined as follows:

$$ MinMaxDist(E, E') = dist(r(E), r(E')) + c(E). $$

A similar definition can be given for the function *MaxMinDist*:

$$MaxMinDist(E, E') = \max\{0, dist(r(E), r(E')) - c(E)\}.$$

Here, the special case that both entries E and E' are identical, i.e. $subtree(E) = subtree(E')$ needs not be considered separately. For example, the maximum distance of an object in an M-tree entry E to its nearest neighbor in E can be conservatively estimated by the covering radius $c(E)$. Figure 4(b) also illustrates all distance approximations for two sample M-Tree entries.

4. METRIC RKNN SEARCH

In the following we will first discuss how the combined pruning strategies and distance approximations can be used for R1NN search, i.e. $k = 1$. After that, we extend these concepts to the general RkNN search, i.e. $k \geq 1$.

4.1 The Basic R1NN Case

Now we want to explore how the distance approximations defined above can be used to identify "true hits" and "true drops" of a R1NN query as early as possible during the traversal of an index $\mathcal{I}$. In contrast to the other distance approximations, the *MaxMinDist* function only plays a role for $k > 1$. The reason for this is that identifying an intermediate entry E with $|subtree(E)| > 1$ as a true hit is obviously not possible for $k = 1$ because we cannot estimate all pairwise distances of the objects in $subtree(E)$ and their distances to q accurately.

Let q be the query object and $E \in \mathcal{I}$ an entry, i.e. an object in $\mathcal{D}$ or an intermediate node of $\mathcal{I}$. Recall that E can be pruned, if for all objects $o \in subtree(E)$, the distance to q is greater than the distance to any other object $o' \neq o$ either in the subtree of E or in the subtree of another index entry $E' \in \mathcal{I}$. Transferred to the problem where only distance approximations are available, entry $E \in \mathcal{I}$ can be pruned, if

$$\exists E' \in \mathcal{I} : MinDist(E, q) > MinMaxDist(E, E').$$

Here, it is assumed that at least one entry, E or E' is an intermediate index node which contains at least two objects within their subtrees. In the case, that both entries E and E' are objects, then we additionally need the condition $E \neq E'$ and $MinMaxDist(E, E') = dist(E, E')$. In order to find an entry $E' \in \mathcal{I}$ which fulfills the above criterion, it suffices to know the minimal $MinMaxDist(E, E')$ from E to all entries $E' \in \mathcal{I}$ which are neither predecessor nor successor of E, i.e. $E \notin subtree(E')$ and $E' \notin subtree(E)$ holds. This minimal $MinMaxDist$, denoted by $MMMDist$ is formalized in the following definition.

DEFINITION 2 (MINIMAL *MinMaxDist*). *Let $\mathcal{I}$ be an index structure and let the entry E be a database object or an intermediate node in $\mathcal{I}$. Furthermore, let $\mathcal{E} := \{E' \in \mathcal{I} \mid E \notin subtree(E') \wedge E' \notin subtree(E)\}$. Then the minimal MinMaxDist of E, denoted by $MMMDist(E)$, is defined as follows:*

$$MMMDist(E) = \min_{E' \in \mathcal{E}} MinMaxDist(E, E').$$

Obviously, $MMMDist(E)$ is (the smallest currently available) upper bound of the 1NN distance of all objects in $subtree(E)$. Thus, if $MMMDist(E) < MinDist(E, q)$ then entry $E \in \mathcal{I}$ can be pruned. On the other hand, we are also able to identify E as true hit but — as discussed above —

only if it is a database object and not an intermediate index entry, i.e. $subtree(E) = e$. In that case, we can compute the exact distance between e and q. The object e is a "true hit", if the following statement holds:

$$\forall E' \in \mathcal{I} : MinDist(e, E') \geq dist(e, q).$$

Therefore, additional to the minimal *MinMaxDist* distance we consider for each object the minimal *MinDist* distance which is defined as follows:

DEFINITION 3 (MINIMAL *MinDist*). *Let $\mathcal{I}$ be an index structure and let the entry e be a database object in $\mathcal{I}$. Furthermore, let $\mathcal{E} := \{E' \in \mathcal{I} \mid e \notin subtree(E')\}$. Then the minimal MinDist of e, denoted by $MMDist(e)$, is defined as follows:*

$$MMDist(e) = \min_{E' \in \mathcal{E}} MinDist(e, E').$$

Obviously, $MMDist(E)$ is (the largest currently available) lower bound of the 1NN distance of e. Thus, e is a true hit if $MMDist(e) \geq dist(e, q)$.

Both the concept of the *MMMDist* and the concept of the *MMDist* give us a starting point to develop an algorithm for R1NN because we just need to keep the *MMMDist* values for all candidate entries up to date during index traversal in order to prune true drops. For candidate objects we can additionally keep the *MMDist* values up-to-date in order to identify true hits. This will enable us to retrieve the final result with only one pass through the index, i.e. we will not need additional refinement rounds where the true 1NN distances of potential candidates are computed. We will discuss the RkNN algorithm which is also valid for the R1NN search problem in more details later.

4.2 The General RkNN Case

The concepts described above for $k = 1$ can be extended to $k \geq 1$. In general, the conversion from $k = 1$ to $k \geq 1$ only requires that the distance to the kNN instead of the 1NN distance has to be taken into account in order to decide whether an object or an intermediate index entry can be pruned or reported as a true hit. However, this has significant impact on the distance estimations based on entry regions. For example, since the *MinMaxDist* and *MaxMinDist* functions relate to a single object of the subtree of a node, it does not suffice any longer to make a worst-case distance estimation w.r.t. the kNNs. Rather, we need other constructs for the approximations of the kNN distances.

The distance approximations which are associated with one or two index entries relate to all objects which belong to the subtrees of the corresponding entries. This means that the distance approximations are related to a set of objects. If we assume that the number of objects a particular distance approximation relates to is known, we can use this information in order to make better estimations of the kNN distance of objects. For that purpose, we exploit the indexing concept as proposed in [8], which allows to store with each index entry E the number of objects covered by E, i.e. the aggregate value $|subtree(E)|$ is stored along with each entry E. For example, the aggregate R-tree (aR-tree) [8, 9] is an instance of this indexing concept. We can assign to each distance approximation between two index entries $E, E' \in \mathcal{I}$ a number of objects the distance approximation relates to. For example, $MinMaxDist(E, E')$ is a distance approximation for each object in $subtree(E)$ and relates to exactly one object

Table 1: The $count(d(E,E'))$ values for the used distance approximations.

d(.,.)	$E = E'$	$E \neq E'$
$MinDist(E,E')$	$\lvert subtree(E') \rvert - 2$	$\lvert subtree(E') \rvert - 1$
$MaxMinDist(E,E')$	1	1
$MaxDist(E,E')$	$\lvert subtree(E') \rvert - 2$	$\lvert subtree(E') \rvert - 1$
$MinMaxDist(E,E')$	1	1

in $subtree(E')$. The distance approximation $MaxDist(E,E')$ which is assigned to each object in $subtree(E)$ relates to $\lvert subtree(E') \rvert$ objects. Let $count(d(E,E'))$ denote the number of objects for which a distance approximation $d(E,E')$ relates to. Table 1 specifies the $count()$ values for the distance approximations used here.

The rationale for subtracting one from $\lvert subtree(E') \rvert$ in case of the $MinDist$ and $MaxDist$ function is that the distance of exactly one object in E' is already approximated more precisely by the $MinMaxDist$ and $MaxMinDist$ functions, respectively. If both entries E and E' are identical, then we additionally have to subtract one object from $subtree(E')$ because the distance approximation of an object $o \in subtree(E')$ can only relate to another object $o' \in subtree(E')$ with $o \neq o'$.

The $count$ function associated with the given distance approximations can be used to estimate the kNN distance of an object $o \in \mathcal{D}$ in order to identify "true drops". Similar to the definition of the $MMMDist$ which is used to prune candidates in case of $k = 1$, we define the k-$MMMDist$ which additionally take the number of objects into account, the correspondingly exploited distance approximations $MaxDist$ and $MinMaxDist$ relates to.

DEFINITION 4 (k-TH MINIMAL $MinMaxDist$). Let $\mathcal{I}$ be an index structure and let the entry E be an intermediate entry or a database object in $\mathcal{I}$. Furthermore, let
$\mathcal{E} := \{E' \in \mathcal{I} \mid E \notin subtree(E')\}$
and let $\hat{D} = < d_1, \ldots, d_{2 \cdot \lvert \mathcal{E} \rvert} >$ be a sequence of $2 \cdot \lvert \mathcal{E} \rvert$ distance approximations sorted in ascending order of their distance values, such that for each entry $E' \in \mathcal{E}$ we have $MinMaxDist(E,E') \in \hat{D}$ and $MaxDist(E,E') \in \hat{D}$. Then the k-th minimal $MinMaxDist$ of E is defined as

$$k\text{-}MMMDist(E) = d_i \in \hat{D},$$

where d_i is the first distance in $\hat{D}$ (in the order of the entries in $\hat{D}$) such that the following criterion is fulfilled:

$$\sum_{1=j}^{i} count(d_j) \geq k.$$

Obviously, the k-$MMMDist(E)$ distance of an entry $E \in \mathcal{I}$ is an upper bound of the kNN distances of all objects $o \in subtree(E)$, i.e.

$$\forall o \in subtree(E) : kNN\text{-}dist(o) \leq k\text{-}MMMDist(E).$$

Thus, an entry $E \in \mathcal{I}$ can be pruned if k-$MMMDist(E) < MinDist(E,q)$.

The $count$ function associated with the distance approximations $MinDist$ and $MaxMinDist$ can analogously be used to identify true hits by estimating a lower bound of the kNN distance of an object $o \in \mathcal{D}$. This estimation is done by the function k-$MMDist$ which is the extension of the $MMDist$

defined above for $k = 1$. In contrast to the special case of $k = 1$ we are now also able to identify intermediate entries E if $\lvert subtree(E) \rvert \leq k$. Thus, the function k-$MMDist$ is defined for arbitrary index entries, i.e. objects and intermediate entries.

DEFINITION 5 (k-TH MINIMAL $MinDist$). Let $\mathcal{I}$ be an index structure and let the entry E be an intermediate entry or a database object in $\mathcal{I}$. Furthermore, let
$\mathcal{E} := \{E' \in \mathcal{I} \mid E \notin subtree(E')\}$
and let $\hat{D} = < d_1, \ldots, d_{2 \cdot \lvert \mathcal{E} \rvert} >$ be a sequence of $2 \cdot \lvert \mathcal{E} \rvert$ distance approximations sorted in ascending order of their distance values, such that for each entry $E' \in \mathcal{E}$ we have $MinMaxDist(E,E') \in \hat{D}$ and $MaxDist(E,E') \in \hat{D}$. Then the k-th minimal $MinDist$ of E is defined as

$$k\text{-}MMDist(E) = d_i \in \hat{D},$$

where d_i is the first distance in $\hat{D}$ (in the order of the entries in $\hat{D}$) such that the following criterion is fulfilled:

$$\sum_{j=1}^{i} count(d_j) \geq k.$$

Obviously, the k-$MMDist(E)$ distance of an entry $E \in \mathcal{I}$ is a lower bound of the kNN distances of all objects $o \in subtree(E)$, i.e.

$$\forall o \in subtree(E) : kNN\text{-}dist(o) \geq k\text{-}MMDist(E).$$

Thus, an entry $E \in \mathcal{I}$ can be reported as true hit if k-$MMDist(E) \geq MinDist(E,q)$.

Similar to the special case $k = 1$ above, the values k-$MMMDist$ and k-$MMDist$ of all entries can be used to formulate a RkNN search algorithm that enables self-pruning of an entry E at any index level as well as mutual-pruning of one entry E w.r.t. another entry E' at any index level. In addition, the k-$MMDist$ function can be used to identify true hits already for intermediate entries E as long as $\lvert subtree(E) \rvert \leq k$. This is beneficial because it saves I/O intensive disc accesses necessary to refine E.

4.3 The RkNN Search Algorithm

The pseudocode of the algorithm for the RNN_k query is illustrated in Figure 6. First, we initialize a pruning list $prune$, a result list $result$ and a priority queue $queue$ which stores index entries E sorted in ascending order by $MinDist(q,E)$. The priority queue is initialized with the root of the index $\mathcal{I}$. Then we dequeue the first entry E of $queue$. If E is a directory node, then E will be refined. The refinement routine is depicted in Figure 7. Before the entry E is refined we have to check which elements of $queue$ will be affected by the refinement of E, i.e. for which elements the lower-bounding and upper-bounding kNN distance approximations $kMMDist(E)$ and $kMMMDist(E)$ have to be updated. This obviously affects all elements c for which $MinDist(c,E) \leq k\text{-}MMMDist(c)$ holds. These elements are then stored in the list $update$. Next, the child entries E_i of E have to be tested, whether they can be pruned or reported as "true hits" by means of their kNN distance approximations $kMMDist(E_i)$ and $kMMMDist(E_i)$. This is done by the filter function apply_filter(E,q,$prune$,$result$) which returns $true$ if E cannot be filtered and, thus, has to be inserted into $queue$. Furthermore, the kNN distance approximations of the elements of the update list $update$ have to be updated
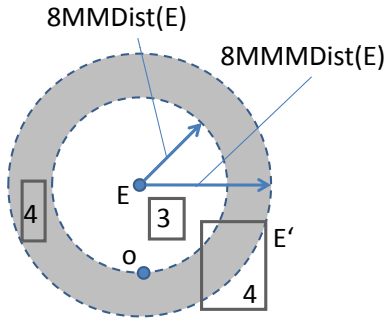
8MMDist(E)

8MMMDist(E)

E
4
3
o
E′
4

Figure 5: Example: Object E cannot be refined anymore, but there must exist at least one another refinement candidate E'.

w.r.t. the child entries of E and the aforementioned filter function has to be applied on them.

If the entry E is a leaf entry, i.e. E is an object, then E obviously cannot be refined. However, if we cannot decide about the status of E as hit or drop yet, we have to get better distance approximations. In fact, in this case there must exist at least one other refinement candidate E' which is a directory entry and which is responsible that E can neither be pruned nor reported as "true hit" by the filter. Such a candidate E' definitely fulfills the following criterion: $MinDist(E, E') \leq k\text{-}MMDist(E)$. Obviously, there might be more than one such candidate. We refine the candidate for which $MinDist(E, E')$ is closest to $k\text{-}MMDist(E)$. This scenario is illustrated in Figure 5. Thereby, the numbers within the directory nodes denotes the number of objects covered by the corresponding node. In our algorithm this candidate is returned by the function $E.k^{th}$-next-entry(). After refining E' we have to update $kMMDist(E)$ and $kMMMDist(E)$ w.r.t. the child entries of E'. Finally we have to check whether E can be filtered or has to be reinserted into *queue*. Note that if E' is also a leaf entry, then the kNN distance approximations of E equal the exact kNN distance. In that case, we can finally decide about the status of E as true hit or true drop. This complete procedure will be repeated until the priority queue has no candidates anymore.

As discussed above, since we keep the kNN distance approximations up-to-date, we implement both the self-pruning and the mutual-pruning paradigm. The algorithm produces the final result in one index traversal without any refinement rounds.

5. EXPERIMENTAL EVALUATION

We compared our novel approach, hereafter referred to as "AKKRZ", for RkNN search with a state-of-the-art approach that do not use any precomputing. For Euclidean data we used the "TPL" algorithm [13] which implements a mutual-pruning strategy. For metric data "TPL" is not applicable. All experiments are based on an R*-Tree for Euclidean data or an M-Tree for metric data with a page size of 4K. Since all approaches are I/O bound we compared the number of disc pages accessed during the execution of 1,000 sample RkNN queries and averaged the results or illustrate the results using box plots that display the mean value, the 25% quantile and the 75% quantile.

5.1 Euclidean Data

```
RkNN(D,q)

   // Initial lists
   prune = EMPTY; result= EMPTY;
   queue = EMPTY; // priority queue sorted by MinDist(q,E).
   queue.add(D.root);

   WHILE (queue is not empty) DO
      E = queue.dequeue();
      IF (E is a DirectoryEntry) THEN refine(E);
      ELSE // E is a LeafEntry
         E′ = E.kth-next-entry(); // see text for description
         refine(E′)
         FOR EACH (E′i ∈ E′) DO
            update k-MMMDist(E, E′i)
            update k-MMDist(E, E′i)
         END-FOR
         IF apply_filter(E,q,prune,result)=true THEN
            queue.insert(E);
         END-IF
      END-IF
   END-WHILE
END RkNN.
```

Figure 6: Pseudocode of the RNN_k algorithm.

```
refine(E,q,queue,prune,result)

   update = {c ∈queue|MinDist(c, E) ≤ k-MMMDist(c)};
   FOR EACH Ei ∈ E DO
      compute MinDist(Ei, q);
      compute kMMMDist(Ei);
      IF apply_filter(E,q,prune,result)=true THEN
         queue.insert(E);
      END-IF
      FOR EACH c ∈ update DO
         update k-MMMDist(c, Ei);
         update k-MMDist(c, Ei);
         IF apply_filter(c,q,prune,result)=false THEN
            queue.remove(c);
         END-IF
      END-FOR
   END-FOR
END refine.
```

Figure 7: Pseudocode of the refinement routine.

We first focus on Euclidean data and compared our AKKRZ algorithm with the TPL algorithm on two synthetic datasets and two real-world datasets. The first synthetic dataset "DS1" contains uniformly distributed 2D points. The second synthetic dataset "DS2" contains clustered 2D points. Both datasets are depicted in Figure 9. The real-world dataset "Genes" contains appr. 1,100 points in a 5D space representing the expression levels of genes. The real-world dataset "Cloud" contains 9D weather parameters recorded at appr. 17,100 different locations in Germany.

Figure 10 displays the performance of the competitors on the four datasets when processing R1NN queries. It can be seen that our novel AKKRZ algorithm significantly 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 11 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

```
apply_filter(E,q,prune,result)
  IF (k-MMMDist(E) < MinDist(E, q)) THEN
      prune.add(E);
  ELSE
    compute k-MMDist(E);
    IF (kMMDist(E) ≥ MinDist(E, q)) THEN
      result.add(E);
    ELSE
      return true;
    ENDIF
  ENDIF
  return false;
END apply_filter_criteria.
```

Figure 8: Pseudocode of the filter function.
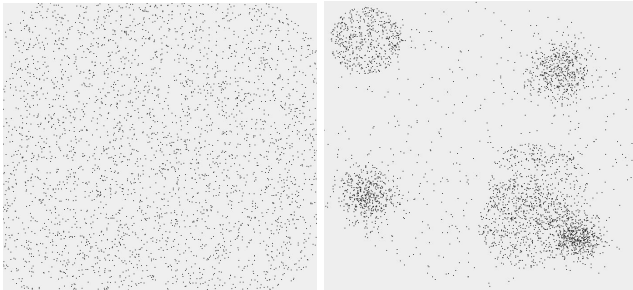
(a) Dataset DS1 with uniformly distributed objects.

(b) Dataset DS2 with clusters.

Figure 9: Synthetic datasets DS1 and DS2.

levels in the index are displayed separately for our AKKRZ approach. As it can be seen, first, the combination of both pruning strategies is beneficial and superior over using only mutual-pruning on the leaf level of the index as done by TPL. In addition, our AKKRZ algorithm can in contrast to the TPL method also mutually prune on higher levels of the index.

Next, we evaluated the scalability of the competitors w.r.t. the number of data objects n. Figure 12 displays the results. Again, the performance gain of our AKKRZ 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.

Last but not least, we evaluated the impact of the query parameter k on the scalability of the competitors. The resulting performances are visualized in Figure 13. It can be observed that our AKKRZ algorithm still clearly outperforms the TPL approach even for higher k values when using the Gene dataset. The reason for this may be that the self-pruning becomes more efficient for this dataset. As can be observed from the experiments on the DS1 dataset, for uniformly distributed data the difference between our approach and the competitor becomes more evident.

5.2 Metric Data

We used the two synthetic datasets "DS1" and "DS2" to evaluate our AKKRZ approach based on the M-tree. Here, the distances are measured by the L_1 norm.
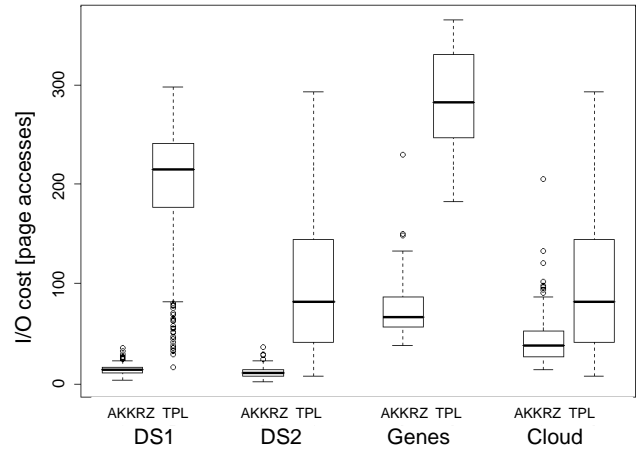

Figure 10: Comparison of AKKRZ and TPL processing R1NN queries on four Euclidean datasets.

Figure 14 displays separately the number of self-pruned objects and the number of mutual-pruned objects for our AKKRZ approach. Similar to the results on Euclidean data the results on metric data show that the combination of both pruning strategies is beneficial and superior over using only self-pruning.

5.3 Summary

In summary, the conducted experiments showed the following. Our novel approach clearly outperformed the competitor even though it solves a more specialized version of the problem. The reason for this is that our approach combines multiple pruning strategies rather than implementing only one pruning paradigm. As a consequence, our new algorithm needs a significantly less number of disc page accesses which in turn means less time to report the results of single RkNN queries.

6. CONCLUSIONS

In recent years, several solutions for the RkNN problem have been proposed. These solutions are limited by specific assumptions. Some assume that the data is of a particular type (Euclidean/general metric) and use certain properties of the data spaces or data type specific data structures. Others assume that the range of possible values for k is limited and that the database remains static, i.e. no updates like insertions or deletions occur. In addition, existing approaches use only one particular pruning strategy, i.e. either self-pruning or mutual-pruning, and, thus, waste possible pruning potentials.

In this paper, we propose a general solution that only assumes that the data is indexed by a tree-like access method which is a rather realistic assumption because the processing of similarity queries without index support is infeasible for real-world database applications. In addition, our solution extends the existing pruning strategies and combines them in order to explore the full pruning potentials. In particular, we presented a generalization of the mutual-pruning strategy from Euclidean spaces to general metric spaces and discussed how it can be used to prune as early as possible during index traversal. Our experimental evaluation shows that our very general solution outperforms the existing specialized
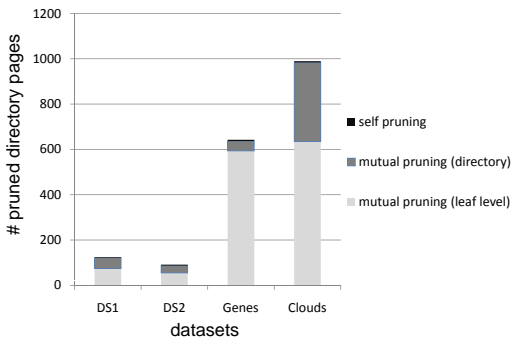
(a) Pruning power w.r.t. directory nodes.
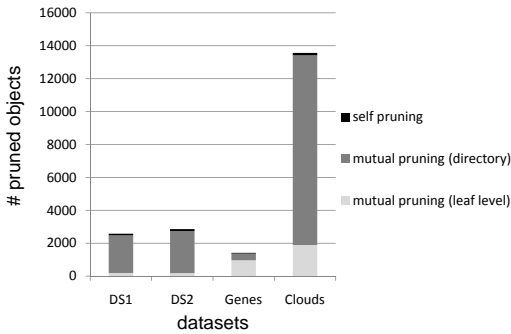
(b) Pruning power w.r.t. data objects.

Figure 11: Benefit of different pruning strategies for AKKRZ.

solutions significantly in terms of query execution times.

For future work, we plan to investigate how our solution can be adapted to databases of continuously moving objects.

7. REFERENCES

[1] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Approximate reverse k-nearest neighbor search in general metric spaces. In *Proc. CIKM*, 2006.

[2] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *Proc. SIGMOD*, 2006.

[3] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor estimation. In *Proc. BTW*, 2007.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD*, pages 322–331, 1990.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-Tree: an efficient access method for similarity search in metric spaces. In *Proc. VLDB*, 1997.

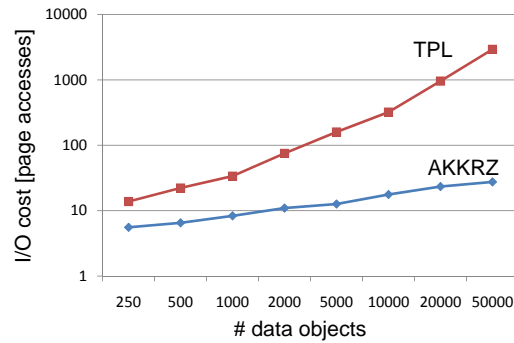[6] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD*, pages 47–57, 1984.

[7] F. Korn and S. Muthukrishnan. Influenced sets based on reverse nearest neighbor queries. In *Proc. SIGMOD*, 2000.

[8] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *Proc. SIGMOD*, 2001.

[9] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *Proc. SSTD*, 2001.

[10] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, San Jose, California, United States*, pages 71–79, 1995.

[11] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High dimensional reverse nearest neighbor queries. In *Proc. CIKM*, 2003.

[12] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *Proc. DMKD*, 2000.

[13] Y. Tao, D. Papadias, and X. Lian. Reverse kNN search in arbitrary dimensionality. In *Proc. VLDB*, 2004.

[14] Y. Tao, M. L. Yiu, and N. Mamoulis. Reverse nearest neighbor search in metric spaces. *IEEE TKDE*, 18(9):1239–1252, 2006.

[15] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. ICDE*, 2001.

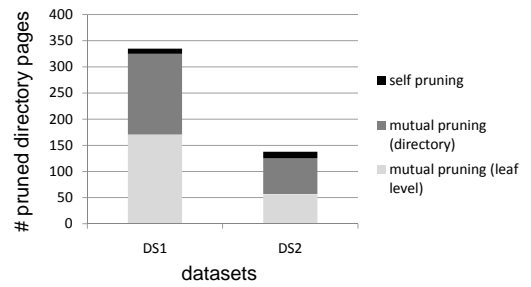**Figure 12: Scalability of the competitors w.r.t. the dataset size.**
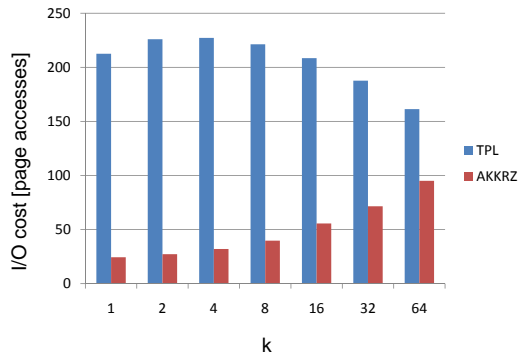
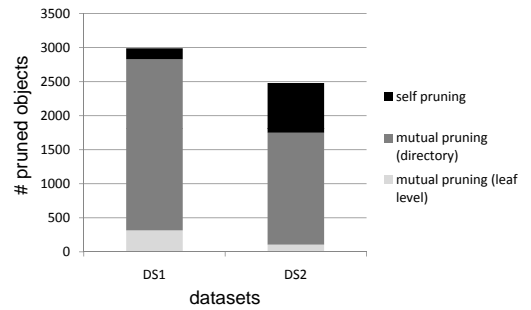(a) DS1.
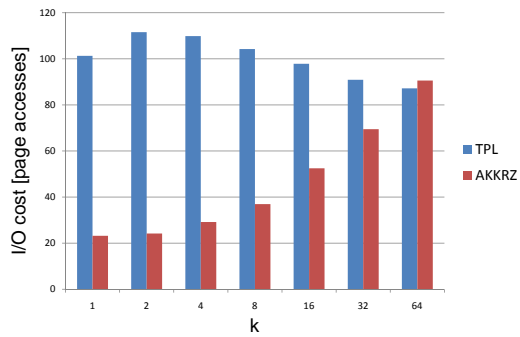
(b) DS2.
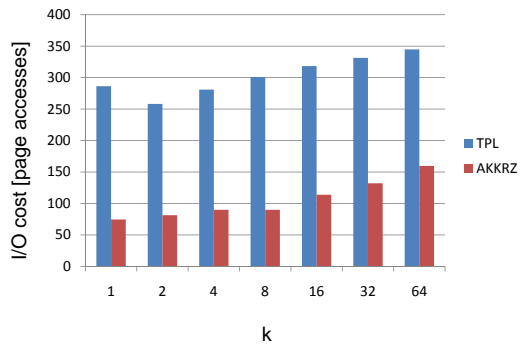
(c) Gene.

Figure 13: Performance of the competitors w.r.t. different values of k.

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

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

Figure 14: Benefit of different pruning strategies for AKKRZ.