Proc. 25th Int. Conf. on Data Engineering (ICDE'09), Shanghai, China, 2009.

Incremental Reverse Nearest Neighbor Ranking

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

Institute for Informatics, Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München, Germany
{kriegel,kroegerp,renz,zuefle,katzdobl}@dbs.ifi.lmu.de

Abstract— **In this paper, we formalize the novel concept of incremental reverse nearest neighbor ranking and suggest an original solution for this problem. We propose an efficient approach for reporting the results incrementally without the need to restart the search from scratch. Our approach can be applied to a multi-dimensional feature database which is hierarchically organized by any R-tree like index structure. Our solution does not assume any preprocessing steps which makes it applicable for dynamic environments where updates of the database frequently occur. Our experiments show that our approach reports the ranking results with much less page accesses than existing approaches designed for traditional reverse nearest neighbor search applied to the ranking problem.**

I. Introduction

While the reverse nearest neighbor (RNN) search problem, i.e. finding all objects in a database that have a given query q among their corresponding k-nearest neighbors, has been studied extensively in the past years, considerably less work has been done so far to support a RNN ranking of objects of a database. An RNN ranking sorts the objects o of the database according to the number of other objects in the database that are more similar to o than q. Thus, if an object o has a ranking score of i w.r.t. a query q, object o would also be a reverse k-nearest neighbor of q for all $k \geq i$ but not a reverse k-nearest neighbor of q for all $k < i$.

Initially, the RNN ranking query reports those objects having the smallest ranking scores in a non-deterministic way since several objects may have the same minimal ranking score. Thereby, the results are reported on demand whenever a function called getNext() is invoked. In other words, each consecutive call of getNext() reports one object with minimal ranking score until there is no unreported object in the database anymore.

The major challenge for algorithms that support rankings in general and RNN rankings in particular is that the result of each getNext()-call should be computed incrementally rather than from scratch, i.e. the current state after each getNext()-call needs to be stored and serves as a starting point to compute the results of the next call. The advantage of an incremental ranking method in general is that no parameter k has to be specified for the query in advance and the first (most relevant) results are reported immediatly without the overhead of simultaneously computing less relevant results. In addition, if the initial results are not sufficient due to any application specific reasons, further results can be requested on demand by calling the getNext() function.

Existing work mainly focused on nearest neighbor ranking [1]. To the best of our knowledge, the task of incremental reverse nearest neighbor ranking has not been considered so far. In this paper, we will formalize this novel ranking problem and provide an efficient solution for it.

The reminder of this paper is organized as follows. In Section II we formally define the RNN ranking problem we want to solve here and discuss related work. Section III explores our novel solution to this problem. In Section IV we present an experimental evaluation. Last but not least, Section V concludes the paper.

II. Survey

A. Problem Formalization

In the following, we assume that $\mathcal{D}$ is a database of n feature vectors 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 kNN distance of q.

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

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

Here, we will be interested in computing a *ranking* of reverse nearest neighbors (RNNs) w.r.t. a query object q rather than in computing the RkNN of q for a fixed value of k. Let the function $R : \mathcal{D} \rightarrow \mathbb{N}$ return for an object $o \in \mathcal{D}$ the number of objects which are closer to o than the query q, i.e. formally,

$$R(o) = |\{p \in \mathcal{D} : dist(p, o) < dist(q, o)\}|.$$

Obviously, it holds that $o \in RNN_k(q)$ iff $R(o) < k$.

The problem of a *reverse nearest neighbor ranking* is to return incrementally all objects $o \in \mathcal{D}$ in increasing order of the values of $R(o)$ by calling the method getNext(). In case of ties, getNext() may report any qualifying object, i.e. we will allow for non-determinism. Let us note that the i-th call of getNext() not necessarily returns an object that is an

[1] The concepts described here can also be extended to any L_p-norm.

RiNN of q, because for a fixed value of k the set $RNN_k(q) - RNN_{k-1}(q)$ generally may contain an (even empty) set of points. In other words, the i-th call of getNext() may report an object o with $R(o) \neq i$. As a consequence, as additional information, the result of each of the ranking steps should include not only the actual object o but also its *ranking count* (*ranking score*) $R(o)$.

B. Related Work

The problem of supporting reverse k-nearest neighbor (RkNN) queries efficiently, i.e. computing for a given query q and a number k the RkNNs of q, has been studied extensively in the past years. Existing approaches for Euclidean RkNN search can be classified as self-pruning approaches or mutual-pruning approaches. *Self-pruning approaches* like the RNN-Tree [4] and the RdNN-Tree [5] are usually designed on top of a hierarchically organized tree-like index structure. They try to estimate the kNN distance of each index entry e. 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 not usually consider other entries (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. 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. *Mutual-pruning approaches* such as [6], [7], [8] use other points to prune a given index entry e. The most general and efficient approach called TPL is presented in [8]. It 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 anymore. 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 o, denoted by $\perp(q, o)$. For the general case, e can be pruned if e is beyond k hyper-planes w.r.t. all current candidates. If e cannot be pruned, it is refined, or, if e is already a database object, e is a new candidate and the hyperplane $\perp(q, e)$ will be considered for pruning in the following. If the ranking queue is empty, the remaining candidate points must be refined, i.e. for each of these candidates, a kNN query must be launched.

Recently, a method for ranked RkNN search has been proposed in [9]. In fact, the authors provide a method for computing the results of an RkNN query with fixed k that are ranked according to k, i.e. the RiNNs are ranked higher than the RjNNs if $i < j \leq k$. This problem is obviously different to the problem of computing an incremental RNN ranking which will be adressed here.

Beside solutions for Euclidean data, solutions for general metric spaces (e.g. [10], [11], [12]) usually implement a self-pruning approach. Typically, metric approaches are less efficient than the approaches tailored for Euclidean data because they cannot make use of the Euclidean geometry.

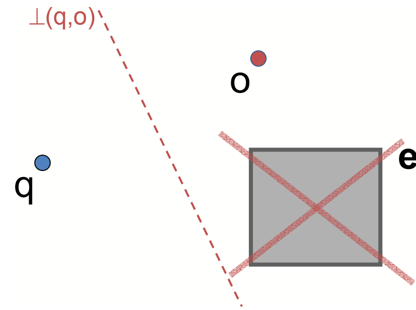


Fig. 1. TPL pruning ($k = 1$).

III. Incremental RNN Ranking

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 like [13], [14], [15]. 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 latter case, *subtree*$(e) = \{e\}$.

The general idea of our solution is based on the TPL-like [8] pruning of entries that are beyond a given number of Voronoi hyperplanes. However, instead of pruning an index entry e, we need to estimate the ranking count value $R(o)$ for all points $o \in$ *subtree*(e). The key observation is that if an index entry e is beyond a Voronoi hyperplane w.r.t. q, then we know that for all $o \in$ *subtree*(e), the value of $R(o)$ can be increased by one. For example, in Figure 1, entry e is beyond the Voronoi hyperplane between q and x, denoted by $\perp(q, x)$. Thus, x will have a smaller distance to all objects $o \in$ *subtree*(e) than q, i.e. all objects $o \in$ *subtree*(e) will have a ranking count $R(o)$ of at least 1. Simply speaking, the ranking count $R(o)$ of any object $o \in \mathcal{D}$ equals the number of Voronoi hyperplanes (including $\perp(q, o)$) that divide the data space such that o and q are in different half spaces.

In the following, we will extend this idea in several important aspects. First, we will extend the concept of Voronoi hyperplanes presented in [8] to higher levels of the index. Originally, the TPL approach considers only Voronoi hyper-planes between the query q and another database object, i.e. at least one leaf entry of the index needs to be fully refined before any Voronoi hyperplane is constructed for pruning. Analogously, this would mean that we can only estimate the ranking count values of objects by means of other objects. This will obviously result in a large overhead of unnecessary page accesses. Rather, we will extend the idea of Voronoi-based pruning/ranking to intermediate entries of the index,

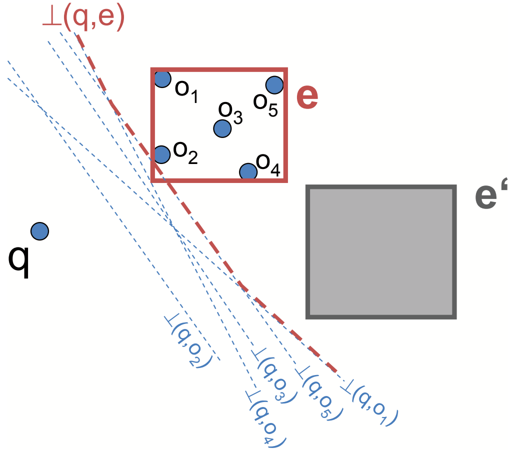


Fig. 2. Conservative approximation $\perp(q, e)$ of the hyperplanes associated with all objects of an index entry e.

i.e. we will also consider Voronoi hyperplanes between the query and intermediate index entries. Second, we will also integrate the idea of self-pruning in order to estimate the ranking count of objects within a given subtree. This will enable us to give better estimations of the ranking counts which will be important for the ranking algorithm. Last but not least, we will present a ranking algorithm based on the two previously mentioned ideas to estimate the ranking count that incrementally computes the next object of an RNN ranking on demand without recomputing the entire ranking from scratch.

A. Ranking Count Estimation

Now we explore strategies for estimating the ranking count based on the hyperplane concept. The basic idea of our approach is to apply the ranking count strategy mentioned above during the traversal of the index, i.e. to identify candidates with high ranking counts as early as possible in order to reduce the I/O costs by saving unnecessary page accesses for the computation of the first results. The ability to push candidates to higher ranking positions already at the directory level of the index implies that a directory entry is used to push itself or other entries.

First, we want to consider the case that a directory entry is used to push other entries back to higher ranking positions by increasing its ranking count. This is similar to the mutual-pruning idea used for RkNN query processing. Generally, the ranking count of an index entry $e \in \mathcal{I}$ can be increased by k according to another entry $e' \in \mathcal{I}$ if there are at least k objects in $subtree(e')$ such that e is behind the Voronoi hyperplane between q and e', denoted by $\perp(q, e')$. In the following a hyperplane *associated with* an entry/object e is denoted by $\perp(q, e)$.

The key idea of the directory-level-wise ranking count estimation 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 o_i in the subtree of e, i.e. $o_i \in subtree(e)$. Figure 2 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, if we exploit the indexing concept as proposed in [2], we also know for the hyperplane associated with that index entry e, $\perp(q, e)$, how many objects this hyperplane relates to. This means that if an entry/object e' is behind a hyperplane $\perp(q, e)$ associated with an index entry e, the entry e' is also behind all hyperplanes $\perp(q, o)$ associated with the objects $o \in subtree(e)$. We can use this information in order to increase the ranking count of entries according to e without accessing the child entries of e. Consequently, the ranking count of an entry/object e' which is behind a hyperplane $\perp(q, e)$ can be increased by $|subtree(e)|$. In Figure 2, the ranking count of entry e' can be increased by 5 because $subtree(e)$ contains five points, i.e. $|subtree(e)| = 5$.

In addition, we can use these considerations also for increasing the ranking count of an intermediate index entry e by itself. This is similar to the self-pruning idea used for RkNN query processing. If an entry $e \in \mathcal{I}$ is behind its own hyperplane, then the ranking count of e can be increased by $|subtree(e)| - 1$, because each object $o \in subtree(e)$ would be behind the hyperplanes associated with all other objects in $subtree(e)$.

B. Ranking Count Updates w.r.t. Intermediate Index Entry Hyperplanes

We are now left with the task to detect when an entry $e' \in \mathcal{I}$ is behind a hyperplane $\perp(q, e)$ associated with an entry $e \in \mathcal{I}$. An important observation is that a hyperplane associated with an object o represents all points p which have the same distance to the query point q and to o, formally:

$$\perp(q, o) = \{p \in \mathbb{R}^d : dist(p, q) = dist(p, o)\}.$$

In addition, we know that all objects stored in the subtree of an index entry e are located inside the minimum bounding hyper-rectangle $e.mbr$ 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. Consequently, the hyperplanes of all objects $o \in subtree(e)$ are conservatively approximated by a hyperplane representation consisting of all points in the vector space that fulfill the following condition:

$$\perp(q, e) = \{p \in \mathbb{R}^d : dist(p, q) = MaxDist(p, e.mbr)\},$$

where $MaxDist(p, e.mbr)$ denotes the maximal distance between the point p and the objects in $e.mbr$, i.e. the distance between p and the most distant corner of $e.mbr$.

In general, a hyperplane representation H is called *conservative approximation* of a set of hyperplanes H', if all objects behind H are definitely behind each hyperplane $h' \in H'$, formally:

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

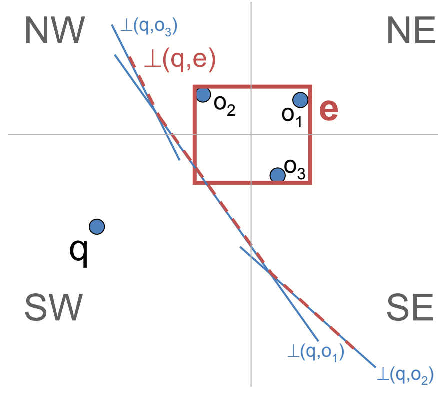


Fig. 3. Computation of conservative hyperplane approximations.

We can assign such a hyperplane representation to each intermediate entry of our index.

In consideration of the above equations, an index entry $e' \in \mathcal{I}$ is defined to be behind a hyperplane $\perp(q, e)$ if the following condition holds:

$$\forall p \in e'.mbr : dist(p, q) > MaxDist(p, e.mbr).$$

Figure 2 illustrates the conservative approximation $\perp(q, e)$ of all hyperplanes $\perp(q, o)$ for all objects $o \in subtree(e)$.

In the following we briefly discuss how this conservative approximation $\perp(q, e)$ can be associated with an index entry e. An important observation is that 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 $subtree(e)$ 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 3 illustrates the computation of such a conservative approximation for a given index entry e in a 2D feature space. First, we have to specify the maximum distance between the 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 $o \in subtree(e)$ which is within the mbr-region of e having the maximum distance to p. This can be done by considering partitions of the vector space which are generated as follows: in each dimension the space is split paraxially at the center of the mbr-region. As illustrated for the 2D example in Figure 3, we obtain partitions denoted by NW, NE, SE and SW. In each of these partitions P, the vertex point of the mbr-region which lies within the diagonal-opposite partition is the mbr-region point which has the maximum distance to all points in P. In our example, for any point p in SW the maximum distance of p to e is the distance between p and point o_1 in partition NE. Consequently, the hyperplane $\perp(q, o_1)$ is a conservative approximation of all hyperplanes between points within the mbr-region of e and the points within the partition SW. In our example, the hyperplane associated with e is composed by the three hyperplanes $\perp(q, o_2)$, $\perp(q, o_1)$ and $\perp(q, o_3)$.

C. Best-First Search Based Incremental RNN Ranking Algorithm

In this section, we show how we explore the index such that the first results can be reported early without causing unnecessary page accesses. We start with an informal description of our solution before we present implementation details and pseudo code.

Similar to the TPL approach for RkNN queries our approach is based on a best-first search method exploiting a priority queue organizing the index entries to be explored. In contrast to the TPL approach, we propose to give the priorities to the index entries according to the estimated ranking count, i.e. entries with low ranking counts are ranked higher than entries with high ranking count. This means that entries containing objects with a low expected ranking position are explored before entries containing objects with a high expected ranking position. The rational for this strategy is that in this way we try to explore those entries first which contain potential candidates to be reported next from the ranking query.

For the organization of the index entries during the traversal of the index we maintain a priority queue $\mathcal{Q}$ storing entries with the corresponding estimated ranking count which are sorted in ascending order according to their estimated ranking count. Thereby we assume that the ranking count of each entry in this queue was generated by taking all current entries in the queue into account using the aforementioned strategies for increasing the ranking count.

The top element of the queue is the entry which has to be explored next. Whenever an entry e is explored, i.e. e is loaded from disk and is refined, we have to perform the following two steps: first, we have to update the ranking counts of all elements in the queue according to the children of e and, second, the ranking counts of e's child elements have to be computed before we insert them into $\mathcal{Q}$.

For the first step, we have to determine those entries in $\mathcal{Q}$ which could be affected by the refinement of e, i.e. for which the ranking count might be increased after refining e. Obviously, those entries which are completely behind the hyperplane representation of e, $\perp(q, e)$, must also be behind the hyperplane representations of each child of e and, thus, their ranking count is not affected by the refinement of e. In the example shown in Figure 4, entry e_3 is not affected by the refinement of entry e due to the above considerations. Furthermore, we can ignore those entries e' which cannot be behind a hyperplane of any object within $subtree(e)$, e.g. entry e_1 in the example in Figure 4, i.e. those entries e' for which the following statement holds:

$$\exists p \in e'.mbr : dist(p, q) < MinDist(p, e.mbr),$$

where $MinDist(p, e.mbr)$ denotes the minimal distance between p and the rectangle $e.mbr$. Intuitively, those entries are

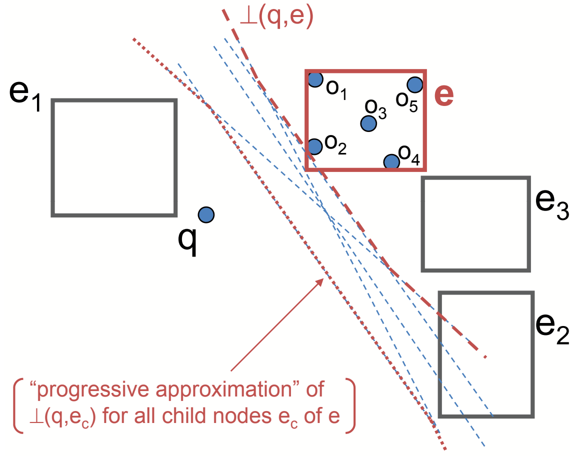


Fig. 4. Illustration of entries that are/are not affected by the refinement of an entry e.

not behind the "progressive approximation" of all hyperplanes $\perp(q, e_c)$ of child entries e_c of entry e (cf. Figure 4).

Entries which are affected by the refinement of e are the remaining entries, i.e. those entries e' that fulfill *both* of the following two conditions:

$$\exists p \in e'.mbr : dist(p, q) \leq MaxDist(p, e.mbr)$$

and

$$\forall p \in e'.mbr : dist(p, q) \geq MinDist(p, e).$$

Each entry e' fulfilling the above two conditions, e.g. entry e_2 in our example in Figure 4, has to be checked against the hyperplane representation of each child of e. If the entry e' is behind the hyperplane representation of a child e_c of e, its ranking counter will be increased by $|subtree(e_c)|$.

For the second step, we have to determine the ranking counts of the children of e. For that purpose, we simply have to check all existing entries in $\mathcal{Q}$ and all other children of e whether the current child e_c of e is behind the corresponding hyperplanes. If yes, the ranking count of e_c is increased by the number of objects included in the subtree of the corresponding entry.

Finally, if the top entry e in the queue $\mathcal{Q}$ is a point, i.e. $e \in \mathcal{D}$, the point can be output as a result only if e is *not* beyond any *progressive* approximation of hyperplanes of child nodes of all e' that are currently in the queue, i.e. formally

$$\forall e' \in \mathcal{Q} : dist(e, q) < MinDist(e, e'.mbr).$$

Otherwise, we need to refine any of those entries $e' \in \mathcal{Q}$, for which this condition does not hold. As a consequence, e might get a higher ranking count and might be shifted towards the end of $\mathcal{Q}$ or it may also maintain the top spot of $\mathcal{Q}$.

The pseudocode of the algorithm for the incremental RNN ranking is illustrated in Figure 5 providing the implementation details of the previously discussed steps. First, we initialize an empty result list "result" and the priority queue $\mathcal{Q}$ which stores index entries sorted in ascending order of their ranking count. Ties occuring in the priority queue are resolved by, first, prefering leaf index entries to directory index entries and, second, by sorting the entries in increasing distance to q.

The priority queue is initialized with the root of the index. For each call of the **getNext** method, we dequeue the first entry e of $\mathcal{Q}$. If e is a directory node, then it will be refined calling the **refine** routine depicted in Figure 6. During refinement, we first have to find all entries in $\mathcal{Q}$ that are candidates for having their ranking count increased due to the refinement of e (see first step above). An entry e' is such a candidate, if there exists a point in the mbr of e' that is closer to q than any point in e (see the predicate in line 3 of the refinement procedure in Figure 6) and if e' has not already been re-ranked by e (see the predicate in line 4 of the refinement procedure in Figure 6). These candidates are stored in a list updateI.

Additionally, we need all entries that are candidates for increasing the ranking count of one of the child entries of e (see the second step above). An entry e'' is such a candidate, if it has not re-ranked e already (first comparison in line 6 of the refinement procedure in Figure 6) and its mbr contains a point that is closer to q than a point in e (second comparison in line 6 of the refinement procedure in Figure 6). These candidates are stored in a list updateII.

Lines 8-12 check for each child node e_c of e and element $e' \in$updateI, if e re-ranks e' and increases the ranking count of e' if necessary. Analogously, lines 13-17 increase the ranking count of e_c, if an element $e'' \in$updateII re-ranks e_c.

Then, we increase the ranking count for each child entry e'_c of e that is able to re-rank e_c. Note that e_c and e'_c may be identical, i.e. e_c re-ranks itself. Finally e_c is inserted into $\mathcal{Q}$.

If the entry e is a leaf entry, i.e. e is an object, then e obviously cannot be refined. However, we may not yet return e as a result without further checking, because it may be re-ranked due to an entry that has not yet been refined. In that case, we need to scan the queue $\mathcal{Q}$ for an object that is a candidate for re-ranking e by calling the **refinementRound** algorithm which is depicted in Figure 7 and refining (c.f. Figure 6) this object. If no such object exists, e can be returned as the result of the current getNext()-call.

IV. Experimental Evaluation

A. Test Bed

We compared our novel approach for computing an RkNN ranking, with two adaptions of the TPL [8] approach which is the current state-of-the-art algorithm for RkNN query processing. In fact, we applied two versions of the TPL approach for computing a ranking. The problem of the TPL approach is that we cannot predict the number of getNext()-calls beforehand. Thus, we do not know a suitable value of k to answer all getNext()-calls.

The first variant, called TPL-Lazy, implements a lazy strategy assuming that we have a low number of getNext()-calls. It manages a result list which is initially empty and a counter k_c which stores the current value of k and is initialized with $k_c = 1$. The entries in the result list are ordered by increasing ranking scores. For each call of the getNext() method, this

```
ALGORITHM  initializeRanking(root, q)
    input: root = root of index storing 𝒟
    input: q = query object
    𝒬 = empty priority queue sorted by RankingCount
    result = ∅
    insert root into 𝒬

METHOD  getNext()
    WHILE 𝒬 is not empty DO
      e = dequeued entry from 𝒬
      IF e is a directory entry THEN
        refine(e, q, 𝒬, result)
      END-IF
      ELSE // e is a LeafEntry
        e' = refinementRound(e, q, 𝒬)
        IF e' = NULL THEN RETURN e
        ELSE refine(e', q, 𝒬, result)
      END-ELSE
    END-WHILE
```

Fig. 5. Pseudocode of the incremental RNN ranking algorithm.

```
METHOD  refine(e, q, 𝒬, result)
    input: e = current directory entry
    input: q = query object
    input: 𝒬 priority queue
    input: result = result list
    updateI= {e' ∈ 𝒬|
        ∀p ∈ e' : MinDist(p, e) ≤ MinDist(p, q)∧
        ∃p ∈ e' : MinDist(p, q) < MaxDist(p, e)}
    updateII= {e'' ∈ (queue ∪ result)|∃p ∈ e :
        MinDist(p, e'') ≤ MinDist(p, q) < MaxDist(p, e'')}
    FOR EACH e_c ∈ e DO
      FOR EACH e' ∈ updateI DO
        IF (∀p ∈ e' : MinDist(p, q) ≥ MaxDist(p, e_c))DO
          increaseRankingCount(e', e_c.weight);
        END-IF
      END-FOR
      FOR EACH e'' ∈ updateII DO
        IF (∀p ∈ e_c : MinDist(p, q) ≥ MaxDist(p, e''))DO
          increaseRankingCount(e_c, e''.weight);
        END-IF
      END-FOR
      FOR EACH e'_c ∈ e DO
        IF (∀p ∈ e_c : MinDist(p, q) ≥ MaxDist(p, e'_c))DO
          increaseRankingCount(e_c, e_c'.weight);
        END-IF
      END-FOR
      queue.insert(e_c)
    END-FOR
```

Fig. 6. Pseudocode of our refine algorithm.

```
METHOD  refinementRound(e, q, 𝒬)
    input: e = current leaf entry
    input: q = query object
    input: 𝒬 priority queue
   FOR EACH entry e' ∈ 𝒬 DO
     IF(MinDist(e, e') ≤ Dist(e, q) < MaxDist(e, e')) THEN
       RETURN e'
     END-IF
    END-FOR
    RETURN NULL
```

Fig. 7. Pseudocode of the refinement round.

variant checks the result list. If the result list is empty, TPL-Lazy computes a RkNN query with $k = k_c$ using the original TPL approach, adds the result of this query to the result list with a ranking score of k_c, and increments k_c. These three steps are processed iteratively until the result list is no longer empty. Last but not least, the TPL-Lazy method returns the next entry in the result list. Obviously, this variant only issues a new RkNN query if necessary beginning with $k = 1$ and successively incrementing the value of k. The costs for answering l getNext()-calls are the sum of the costs of all queries for $k = 1, \ldots$ necessary to answer the l calls.

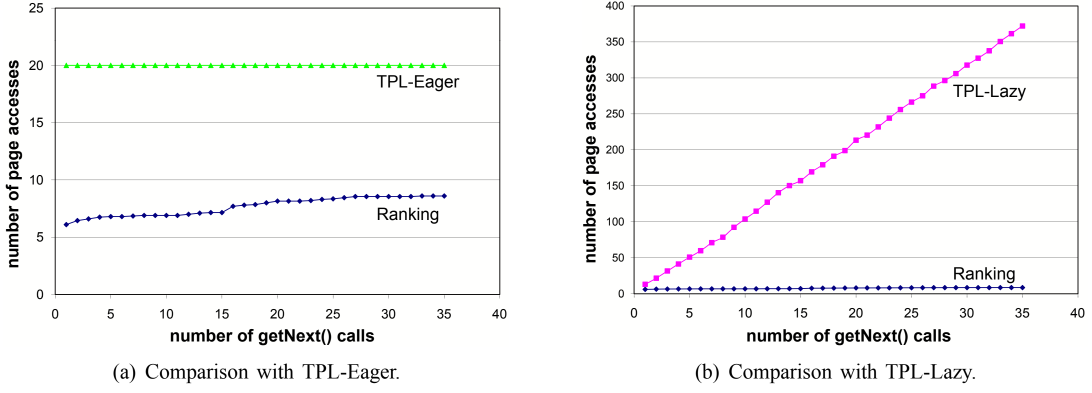
The second variant, called TPL-Eager, implements an eager policy assuming a higher but possible fixed maximum number of getNext()-calls. It simply assumes that the maximum number of getNext()-calls will be less than the number of result objects of a RkNN query with a special value of k_{max}, e.g. $k_{max} = 100$. Then, we only need to issue one Rk_{max}NN query using the original TPL approach beforehand and sort the results according to their ranking score. Whenever a getNext()-call is issued (and as long as the assumptions stated above regarding the size of the result and the number of getNext()-calls hold), we can simply return the next object from the result list. The costs for answering l getNext()-calls equal to the costs of answering the Rk_{max}NN query (again, as long as the result contains at least l points).

Let us note that there is no direct relationship between the number of getNext()-calls l and the value k_{max}. This makes it even harder for the TPL-Eager approach to guess a proper k_{max} value. In fact, to obtain a fair comparison, we computed the most optimistic scenario for the TPL-Eager variant: we first issued l getNext()-calls with our new ranking method and obtained the ranking count of the resulting point of the last call. This count is the optimal k_{max} value for the TPL-Eager approach and we used this value in all our experiments. Thus, in realistic scenarios, the results of a TPL-Eager approach would be worse than presented here.

All experiments are based on an aR*-Tree (aggregate version of R*-Tree) with a page size of 1K. Since all approaches are I/O bound we compared the number of disc pages accessed during the execution of 500 sample RkNN queries and averaged the results.

B. Synthetic Data

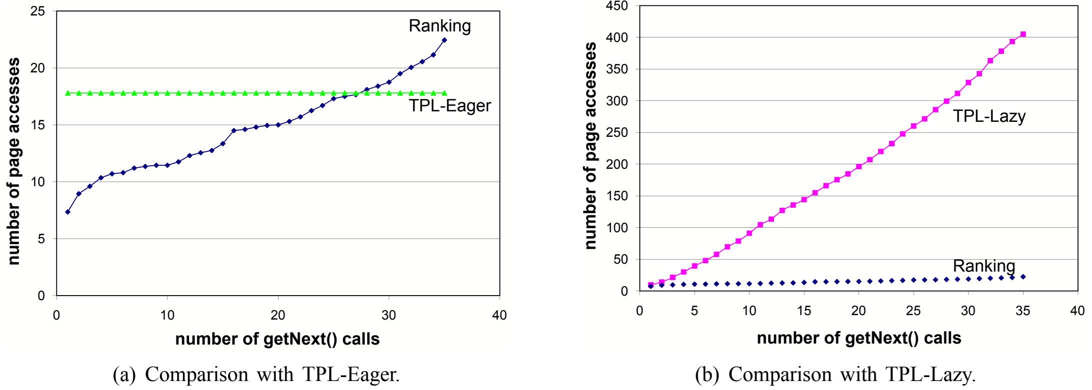
We used two synthetic datasets to compare the performance of our ranking algorithm with the two variants of TPL. The first dataset contains 10,000 uniformly distributed 2D points. Figure 8 displays the performance of the competitors w.r.t. the number of getNext()-calls. As expected, the performance of the TPL-Eager approach (c.f. Figure 8(a))is constant as long as the number of getNext()-calls is smaller than the number of results of the Rk_{max}NN query issued beforehand (which is the case in our scenario – see above). Nevertheless, our ranking algorithm clearly outperforms this TPL variant in terms of query execution times. In fact, the costs of our approach increase only slightly with successive getNext()-calls. In addition, it should be noted that TPL-Eager would need to issue a new RkNN query with a considerably higher value of k if we have more than 35 getNext()-calls because



(a) Comparison with TPL-Eager.

(b) Comparison with TPL-Lazy.

Fig. 8. Comparison of the novel RkNN ranking with different extensions of TPL on the uniform 2D synthetic dataset.



(a) Comparison with TPL-Eager.

(b) Comparison with TPL-Lazy.

Fig. 9. Comparison of the novel RkNN ranking with different extensions of TPL on the clustered 2D synthetic dataset.

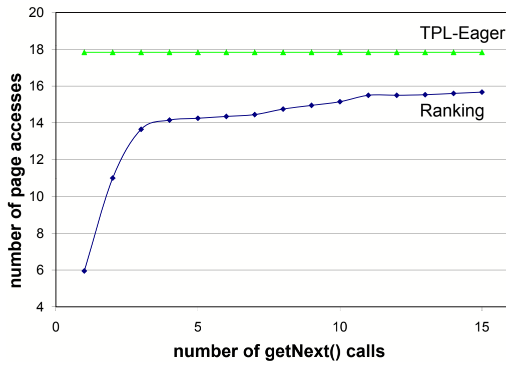
TPL-Eager was optimized for 35 results. Thus, in that case, we would have a jump for the TPL-Eager approach at the 36th getNext()-call while the costs of our ranking algorithm will most likely evolve like in the range of the first 35 getNext()-calls. On the other hand, the costs for the TPL-Lazy variant (cf. Figure 8(b)) increase much faster than the costs of our new ranking algorithm. Again our approach clearly outperforms the competitor in terms of query execution times. Note that the performance of our ranking algorithm is of course the same in both Figures 8(a) and 8(b).

A similar observation can be obtained from Figure 9 which displays the performance of the competitors on a 2D synthetic dataset that contains 10,000 points clustered into four different clusters. The only obvious difference is that here, the TPL-Eager approach performs much better than on the uniform dataset. As illustrated in Figure 9(a), our ranking algorithm outperforms the TPL-Eager variant only for the first 27 getNext()-calls. In this setting, the TPL-Eager slightly outperforms our ranking algorithm for 30 to 35 getNext()-calls. However, please note that, first, the TPL-Eager approach was implemented with the most optimistic assumptions and can be expected to perform considerably worse in a more realistic scenario where the perfect k_{max} value can usually not be determined beforehand. Second, as explained above, TPL-Eager was optimized for 35 results. If we had more than 35 getNext()-calls, then TPL-Eager would be required again to compute a new RkNN query with a considerably higher value of k which would cause significantly higher costs from the 36th getNext()-call on until the next jump limit is reached.

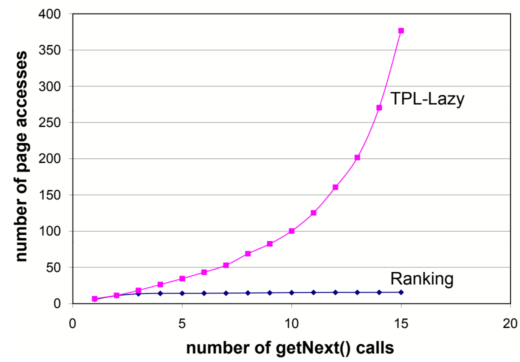
On the other hand, in comparison to the TPL-Lazy variant (cf. Figure 9(b)) our ranking algorithm again performs much better and significantly outperforms the competitor in terms of query execution times. Again, the performance of our ranking algorithm is of course the same in both Figures 9(a) and 9(b).

C. *Real-world Data*

We also tested our novel ranking algorithm on real-world data. In Figure 10 the performance of our ranking algorithm is compared with the performances of the TPL-Eager variant (cf. 10(a)) and the TPL-Lazy variant (cf. 10(b)) on a dataset that features the expression level of approx. 6,000 genes under 5 conditions. The result on this 5D dataset is similar to the results on the synthetic datasets reported above. Again, our ranking algorithm clearly outperforms the TPL-Lazy approach. Analogously, the difference to the TPL-Eager approach is less significant but still considerable. It should again be noted that the TPL-Eager variant assumes the most optimistic scenario for its application which is most likely not a realistic setting, and, thus, it can be expected that TPL-Eager performs less accurate in most applications.



(a) Comparison with TPL-Eager.



(b) Comparison with TPL-Lazy.

Fig. 10. Comparison of the novel RkNN ranking with different extensions of TPL on the 5D gene expression dataset.

D. Summary

To summarize the results of our experimental evaluation, our novel ranking algorithm outperforms both adaptions of the existing TPL algorithm to the ranking problem, TPL-Eager and TPL-Lazy, significantly in terms of query execution times. While TPL-Eager seems to be competitive (if at all) only for a higher number of getNext()-calls, TPL-Lazy seems to be competitive (if at all) only for a very low number of getNext()-calls. This result is quite intuitive because TPL-Eager tries to estimate the worst-case by precomputing the maximum number of required results for a maximum number of getNext()-calls by computing one Rk_{max}NN query. Thus, the more the number of getNext()-calls reaches the number of resulting objects of the Rk_{max}NN query, the more the costs for the Rk_{max}NN query pay off. Otherwise, TPL-Eager caused a large portion of unnecessary costs to compute a large number of results that are not needed. On the other hand, TPL-Lazy assumes the best case of very few getNext()-calls and, thus, computes results only if necessary by consecutively issuing a RkNN query with increasing k. Obviously, as long as the number of consecutive RkNN queries, with increasing k necessary to report results, is small, i.e. the number of getNext()-calls is low, this strategy pays off. Otherwise, TPL-Lazy constantly recomputes RkNN queries with the next higher value for k which produces a lot of redundant results w.r.t. the previously computed queries.

Our ranking algorithm obviously performs best because it does not assume worst- or best-cases but focuses on computing the ranking incrementally. Since in a ranking query scenario, it is not known beforehand, how often the method getNext() is called, this is the most efficient solution in the general case but also – as our experiments illustrate – in the borderline cases where either TPL-Eager or TPL-Lazy perform best.

V. Conclusions

In this paper, we formalize a novel ranking problem, the reverse nearest neighbor (RNN) ranking and propose an original solution for it. Our solution extends existing methods for RNN query processing in the following important aspects. First, the mutual-pruning strategy of existing approaches is generalized and adapted so that it can be applied already on higher levels of the index and it can be applied to estimate the ranks of an index entry rather than for pruning. Second, we also incorporated the idea of self-pruning and explored how this concept can be applied to estimate the ranking of index entries. Last but not least, we proposed an incremental algorithm for the RNN ranking problem that is based on both introduced ranking estimations. Our experimental evaluation confirms that our new solution outperforms existing methods adapted for the new problem significantly in terms of query execution times.

References

[1] G. R. Hjaltason and H. Samet, "Ranking in spatial databases," in *Proc. SSD*, 1995.
[2] I. Lazaridis and S. Mehrotra, "Progressive approximate aggregate queries with a multi-resolution tree structure," in *Proc. SIGMOD*, 2001.
[3] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, "Efficient olap operations in spatial data warehouses," in *Proc. SSTD*, 2001.
[4] F. Korn and S. Muthukrishnan, "Influenced sets based on reverse nearest neighbor queries," in *Proc. SIGMOD*, 2000.
[5] C. Yang and K.-I. Lin, "An index structure for efficient reverse nearest neighbor queries," in *Proc. ICDE*, 2001.
[6] I. Stanoi, D. Agrawal, and A. E. Abbadi, "Reverse nearest neighbor queries for dynamic databases," in *Proc. DMKD*, 2000.
[7] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun, "High dimensional reverse nearest neighbor queries," in *Proc. CIKM*, 2003.
[8] Y. Tao, D. Papadias, and X. Lian, "Reverse kNN search in arbitrary dimensionality," in *Proc. VLDB*, 2004.
[9] K. C. K. Lee, B. Zheng, and W.-C. Lee, "Ranked reverse nearest neighbor search," *IEEE TKDE*, vol. 20, no. 7, pp. 894–910, 2008.
[10] 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.
[11] ——, "Approximate reverse k-nearest neighbor search in general metric spaces," in *Proc. CIKM*, 2006.
[12] Y. Tao, M. L. Yiu, and N. Mamoulis, "Reverse nearest neighbor search in metric spaces," *IEEE TKDE*, vol. 18, no. 9, pp. 1239–1252, 2006.
[13] A. Guttman, "R-Trees: A dynamic index structure for spatial searching," in *Proc. SIGMOD*, 1984, pp. 47–57.
[14] 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*, 1990, pp. 322–331.
[15] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-Tree: An index structure for high-dimensional data," in *Proc. VLDB*, 1996.