A Mutual Pruning Approach for RkNN Join Processing

Tobias Emrich, Peer Kröger, Johannes Niedermayer, Matthias Renz, Andreas Züfle

Institute for Informatics, Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, Germany
{emrich,kroeger,niedermayer,renz,zuefle}@dbs.ifi.lmu.de

Abstract: A reverse $k$-nearest neighbour (R$k$NN) query determines the objects from a database that have the query as one of their $k$-nearest neighbors. Processing such a query has received plenty of attention in research. However, the effect of running multiple R$k$NN queries at once (join) or within a short time interval (bulk/group query) has, to the best of our knowledge, not been addressed so far. In this paper, we analyze R$k$NN joins and discuss possible solutions for solving this problem. During our performance analysis we provide evaluation results showing the IO and CPU performance of the compared algorithms for a variety of different setups.

1 Introduction

A Reverse $k$-Nearest Neighbor (R$k$NN) query retrieves all objects from a database having a given query object as one of their $k$ nearest neighbors. Various algorithms for efficient R$k$NN query processing have been studied under different conditions due to the query’s relevance in a wide variety of domains — applications include decision support, profile-based marketing and similarity updates in spatial and multimedia databases.

Let us now shortly recap the definition of R$k$NN queries. Given a finite multidimensional data set $S \subset \mathbb{R}^d$ ($s_i \in \mathbb{R}^d$), a query point $r \in \mathbb{R}^d$, and an arbitrary distance function $dist(x, y)$ (e.g. the Euclidean distance), a $k$-nearest neighbor ($k$NN) query returns the $k$ nearest neighbors of $r$ in $S$:

$$kNN(r, S) = \{ s \in S : |\{ s' \in S : dist(s', r) < dist(s, r)\}| < k \}$$

A monochromatic R$k$NN query, where $r$ and $s \in S$ have the same type, can be defined by employing the $k$NN query:

$$RkNN(r, S) = \{ s \in S | r \in (k+1)NN(s, S \cup \{r\}) \}$$

Thus, an R$k$NN query returns all points $s_i \in S$ that would have $r$ as one of its nearest neighbors. In Figure 1(a) an R2NN query is shown. Arrows denote a subset of the 2NN relationships between points from $S$. Since $r$ is closer to $s_2$ than its 2NN $s_1$, the result set of an R2NN query with query point $r$ is $\{s_2\}$. $s_3$ is not a result of the query since its 2NN $s_2$ is closer than $r$. Note that the R$k$NN query is not symmetric, i.e. the $k$NN result $kNN(r, S) \neq RkNN(r, S)$, because the 2NN of $r$ are $s_2$ and $s_3$. Therefore the result of an R$k$NN($r, S$) query cannot be directly inferred from the result of a $k$NN query $kNN(r, S)$.

Besides the monochromatic R$k$NN query, research often discusses the bichromatic R$k$NN query. However, in this paper, we will concentrate on the monochromatic case and will
therefore just shortly introduce this second variant of the \(R_k\)NN query. In the bichromatic case, two sets \(R_1\) and \(R_2\) are given. The goal is to compute all points in \(R_2\) for which a query point \(r \in R_1\) is one of the \(k\) closest points from \(R_1\) \([\text{WYCT08}]\):

\[
BR_kNN(r, R_1, R_2) = \{ s \in R_2 | r \in kNN(s, R_1) \}
\]

An important problem in database environments is the scenario where the query does not consist of a single point but instead of a whole set of points, for each of which a \(R_k\)NN query has to be performed. This setting is often referred to as group query, bulk query or simply join of two sets \(R\) and \(S\). Despite the potential applications, the join operation has so far only received little attention in the context of \(R_k\)NN queries. Given two sets \(R\) and \(S\), the goal of a monochromatic \(R_k\)NN join is to compute, for each point \(r \in R\) its monochromatic \(R_k\)NNs in \(S\).

**Definition 1 (Monochromatic \(R_k\)NN join)** Given two finite sets \(S \subset \mathbb{R}^d\) and \(R \subset \mathbb{R}^d\), the monochromatic \(R_k\)NN join \(R \bowtie S\) returns a set of pairs containing for each \(r \in R\) its \(R_k\)NN from \(S\): \(R \bowtie S = \{ (r, s) | r \in R \land s \in S \land s \in R_kNN(r, S) \}\)

An example for \(k = 1\) can be found in Figure 1(b). The result for both objects from \(R\) in this example is \(R1NN(r_1) = R1NN(r_2) = \{ s_2 \}\), i.e. \(R \bowtie S = \{ (r_1, s_2), (r_2, s_2) \}\). Note that the elements \(r_1\) and \(r_2\) from \(R\) do not influence each other, i.e., \(r_1\) cannot be a result object of \(r_2\) and vice versa. This follows directly from the definition of the \(MRk\)NN join.

In this paper we discuss two solutions for solving \(R_k\)NN joins. The first solution simply involves the iterative execution of an existing algorithm, while for the second solution we introduce an algorithm specialized for \(R_k\)NN joins. The resulting algorithms are evaluated in an experimental section under a variety of different setups, including both synthetic and real data sets.

The remainder of this paper is organized as follows. Section 2 gives an insight into related work. In Section 3 we propose an \(R_k\)NN join algorithm that is based on an existing mutual pruning algorithm. An extensive performance comparison of our solution follows in Section 4. Section 5 concludes this work.
2 Related Work

The problem of efficiently supporting R$k$NN queries has been studied extensively in the past years. Existing approaches for Euclidean R$k$NN search can be classified as self pruning approaches or mutual pruning approaches. *Self pruning approaches* [KM00, YL01, ABK+06b, TYM06] are usually designed on top of a hierarchically organized tree-like index structure. They try to conservatively/exactly estimate the $k$NN distance of each index entry $e$. If this estimate 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 $k$NN distance of an entry $e$, but simply precompute $k$NN distances of database points and propagate these distances to higher level index nodes.

*Mutual pruning approaches* such as [SAA00, SFT03, TPL04] use other points to prune a given index entry $e$. The most general and efficient approach called TPL is presented in [TPL04]. We will employ this approach as a benchmark algorithm during our performance evaluation.

The approach of combining self- and mutual pruning has been followed in [AKK+09, KKR+09b]. It obtains conservative and progressive distance approximations between a query point and arbitrarily approximated regions of a metric index structure.

Beside solutions for Euclidean data, solutions for general metric spaces (e.g. [ABK+06b, ABK+06a, TYM06]) usually implement a self pruning approach.

Furthermore, there exist approximate solutions for the R$k$NN query problem that aim at reducing the query execution time for the cost of accuracy [SFT03, XHL+05].

Besides the attention paid to single R$k$NN queries, the problem of performing multiple R$k$NN queries at a time, i.e. a R$k$NN join, has hardly been addressed. The authors of [YZHX10] addressed incremental bichromatic R$k$NN joins as a by-product of incremental $k$NN joins, aiming at maintaining a result set over time instead of performing bulk evaluation of large sets. Since it does not address the problem of a monochromatic join, it solves a different problem.

3 The Mutual Pruning Algorithm

Mutual pruning approaches such as TPL [TPL04] are state-of-the-art solutions for single R$k$NN queries. In this paper we aim at analyzing whether this assumption still holds for an R$k$NN join setting. Therefore, in this section, we propose an algorithm for processing R$k$NN joins based on a mutual pruning strategy similar to TPL. We assume that both sets $R$ and $S$ are indexed by an aggregated hierarchical tree-like access structure such as the $aR^*$-tree [PKZT01]. An $aR^*$-Tree is equivalent to an $R^*$-Tree but stores an additional integer value (often called weight) within each entry, corresponding to the number of objects contained in the subtree. The indexes are denoted by $R$ and $S$, respectively.
3.1 General Idea

The proposed algorithm is based on a solution for Ranking-RkNN queries, initially suggested in [KKR+09a]. Unlike TPL, which can only use leaf entries (points) to prune other leaf entries and intermediate entries (MBRs), the technique of [KKR+09a] further permits to use intermediate entries for pruning, thus, allowing to prune entries while traversing the tree, without having to wait for \( k \) leaf entries to be refined first. The algorithm of [KKR+09a] uses the MAXDIST-MINDIST-approach as a simple method for mutual pruning using rectangles. This approach exploits that, for three rectangles \( R, A, B \), it holds that \( A \) must be closer to \( R \) than \( B \), if \( \maxDist(A, R) < \minDist(B, R) \). The algorithm that we use in this work, will augment the algorithm of [KKR+09a] by replacing the MAXDIST-MINDIST-approach by the spatial pruning approach proposed in [EKK+10] which is known to be more selective. In the following, the base algorithm of [KKR+09a], enhanced by [EKK+10] will be extended to process joins.

The mutual pruning approach introduced in this section is based on an idea which is often used for efficient spatial join processing: Both indexes \( R \) and \( S \) are traversed in parallel, result candidates for points \( r \in R \) of the outer set are collected and for each point \( r \in R \) irrelevant subtrees of the inner index \( S \) are pruned; we will evaluate if this approach is also useful for RkNN joins during performance analysis. Thus, at some point of traversing both trees, we will need to identify pairs of entries \((e_R \in R, e_S \in S)\) for which we can already decide, that for any pair of points \((r \in e_R, s \in e_S)\) it must/must not hold that \( s \) is a RkNN of \( r \). To make this decision without accessing the exact positions of children of \( e_R \) and \( e_S \), we will use the concept of spatial domination ([EKK+10]): If an entry \( e_R \) is (spatially) dominated by at least \( k \) entries in \( S \) with respect to \( e_S \), then no point in \( e_S \) can possibly have any point of \( e_R \) as one of its \( k \) nearest neighbors. Due to the spatial extend of MBRs, this decision is not always definite. We have to distinct several cases, as illustrated in Figure 2. The subfigures visualize two pages \( e_R^0 \) and \( e_S^0 \), and one of the additional pages \( e_1^S, e_2^S, e_3^S \). The striped areas in the picture denote the set of points on which a closer decision can definitely be made. This means, no matter which points from the rectangle \( e_R^0 \) and \( e_S^0 \) are chosen, a point in the striped area is always closer to the point from \( e_R^0 \) (or \( e_S^0 \)) than to the point from \( e_S^0 \) (or \( e_R^0 \)). Therefore, in the first case, \( e_1^S \) is definitely closer to \( e_0^S \) than to \( e_R^0 \). In the second case, \( e_2^S \) is definitely closer to \( e_R^0 \) than to \( e_0^S \). In the third case, in all of the four subcases, no decision can be made.
More formally, in the first case, we can decide that an entry is (spatially) dominated by another entry. For example, in Figure 2(a), entry $e^R$ is dominated by entry $e^S_1$ with respect to entry $e^S_0$, since for all possible triples of points $(s_0 \in e^S_0, s_1 \in e^S_1, r \in e^R)$ it holds that $s_1$ must be closer to $s_0$ than $r$. This domination relation can be used to prune $e^S_1$.

If the number of objects contained in $e^S_1$ is at least $k$, then we can safely conclude that at least $k$ objects must be closer to any point in $e^S_0$, and, thus, $e^S_0$ and all its child entries can be pruned. To efficiently decide if an entry $e^S_1$ dominates an entry $e^R$ with respect to an entry $e^S_0$ (all entries can be points or rectangles), we utilize the decision criterion $\text{Dom}(e^S_1, e^R, e^S_0)$ proposed in [EKK+10] which prevents us from doing a costly materialization of the pruning regions like the striped areas in Figure 2. Materialization here means the exact polygonal computation of the areas that allow pruning

In the second case, we can decide that neither an entry, nor its children can possibly be pruned by another entry. In Figure 2(b), consider entry $e^S_2$. It holds that for any triple of points $(s_0 \in e^S_0, s_2 \in e^S_2, r \in e^R)$, that $s_2$ cannot be closer to $s_0$ than $r$. Although, in this case, we cannot prune $e^S_2$, we can safely avoid further domination tests of children of the tested entries. We can efficiently perform this test by evaluating the aforementioned criterion $\text{Dom}(e^R, e^S_2, e^S_0)$.

Finally, in the third case, both predicates $\text{Dom}(e^S_1, e^R, e^S_0)$ and $\text{Dom}(e^R, e^S_2, e^S_0)$ do not hold for any entry $e^S_3$ in Figure 2(c). In this case, some points in $e^S_3$ may be closer to some points in $e^R$, while other points may not. Thus, we have to refine at least some of the entries $e^S_0, e^S_1$, or $e^R$. The reason for the inability to make a decision here, is that the pruning region between two rectangles is not a single line, but a whole region (called tube here, cf. Figure 2). For objects that fall into the tube, no decision can be made.

At any time of the execution of the algorithm only one entry $e^R$ of the outer set is considered. For $e^R$, we minimize the number of domination checks that have to be performed. Therefore, we keep track of pairs of entries in $S$, for which case three holds, because only in this case, refinement of entries may allow to prune further result pairs. This is achieved by managing, for each entry $e^S \in S$, two lists $e^S,.update1 \subset S$ and $e^S,.update2 \subset S$: List $e^S,.update1$ contains the set of entries with respect to which $e^S$ may dominate $e^R$ but does not dominate $e^R$ for sure. Essentially, any entry in $e^S,.update1$ may be pruned if $e^S$ is refined. List $e^S,.update2$ contains the set of entries, which may dominate $e^R$ with respect to $e^S$, but which do not dominate $e^R$ for sure. Thus, $e^S,.update2$ contains the set of entries, whose children may potentially cause $e^S$ to be pruned.

3.2 The Algorithm $\text{joinEntry}$

In order to implement these ideas, we use the recursive function shown in Algorithm 1, $\text{joinEntry}(Entry e^R, Queue Q^S)$. It receives an entry $e^R \in R$ that represents the currently processed entry from the index of the outer set $R$, which can be a point, a leaf node containing several points, or an intermediate node. $Q^S$ represents a set of entries from $S$ sorted decreasingly in the number $|e^S,.update1|$ of objects that an entry $e^S \in S$ is able to prune. The reason is that resolving nodes with a large $update1$ list potentially allows pruning many other nodes.
Algorithm 1 joinEntry(Entry $e_R$, Queue $Q^S$)

1: for all $e^S_i \in Q^S$ do
2:   {Update domination count (lower bound) of all $e^S_i$}
3: for all $e^S_j \in e^S_i.update2$ do
4:   if $\text{Dom}(e^S_j, e^R_i, e^S_i)$ then
5:     {definite decision possible, $e^S_j$ prunes $e^S_i$}
6:     $e^S_i.dominationCount += e^S_j.weight$
7:   else if $\text{Dom}(e^R, e^S_j, e^S_i)$ then
8:     {$e^S_i$ can definitely not be pruned by $e^S_j$}
9:     $e^S_i.update2.remove(e^S_j)$
10:    $e^S_j.update1.remove(e^S_i)$
11:   end if
12: end for
13: if $e^S_i.dominationCount \geq k$ then
14:   {no point in $e^S_i$ can be an R$k$NN of a point in $e^R$}
15:   delete($Q^S, e^S_i$)
16: end if
17: end for
18: {in the following, resolve $S$}
19: Queue $Q^S = \emptyset$
20: while ($e^S_i = Q^S.poll() \neq \text{NULL}$) do
21:   Go to line 20 if $e^S_i.dominationCount \geq k$ { does not contain result candidates}
22:   if Vol($e^S_i$) > Vol($e^R$) then
23:     {go one level down in the subtree of $e^S_i$ and add child pages to $Q^S$}
24:     $Q^S.add(\text{resolve($e^S_i, e^R$)})$
25:   else if isLeaf($e^S_i$) ∧ isLeaf($e^R$) then
26:     {if no further refinement is possible, results still have to be verified}
27:     if $e^R \in kNN(e^S_i)$ then
28:       reportResult($<e^R, e^S_i>$)
29:     end if
30:   else
31:     {put pages $e^S_i$ into $Q^S_c$ if they could neither be pruned nor reported as result}
32:     $Q^S_c.add(e^S_i)$
33:   end if
34: end while
35: {in the following, resolve $e^R$}
36: if ¬isLeaf($e^R$) then
37:   {finally, refine $e^R_i$ by recursively calling joinEntry with $Q^S_c$}
38:   for all $e^R_i \in e^R_i.children$ do
39:     joinEntry($e^R_i$, clone($Q^S_c$))
40:   end for
41: end if
Algorithm 2 resolve(Entry \(e^S\), Entry \(e^R\))

1: LIST l
2: \{ (1) check which objects the children \(e^S_i\) of \(e^S\) may affect \}
3: for all \(e^S_j \in e^S\).children do
4: \(e^S_j\).update.remove\((e^S)\) \{ remove, children of \(e^S\) are now relevant instead of \(e^S\) \}
5: for all \(e^S_j \in e^S\).children do
6: if Dom(\(e^S_i, e^R, e^S_j\)) then
7: \{ definite decision possible, \(e^S_i\) prunes \(e^S_j\) \}
8: \(e^S_j\).dominationCount += \(e^S_i\).weight
9: else if ¬ Dom(\(e^R, e^S_i, e^S_j\)) then
10: \{ no definite decision possible, \(e^S_i\) might prune \(e^S_j\) \}
11: \(e^S_j\).update2.add\((e^S_i)\)
12: \(e^S_i\).update1.add\((e^S_j)\)
13: end if
14: end for
15: end for
16: \{ (2) check which other entries may affect a child \(e^S_i\) \}
17: for all \(e^S_j \in e^S\).children do
18: for all \(e^S_j \in e^S\).update2 do
19: if Dom(\(e^S_j, e^R, e^S_i\)) then
20: \{ definite decision possible, \(e^S_j\) prunes \(e^S_i\) \}
21: \(e^S_i\).dominationCount += \(e^S_j\).weight
22: else if ¬ Dom(\(e^R, e^S_j, e^S_i\)) then
23: \{ no definite decision possible, \(e^S_j\) might prune \(e^S_i\) \}
24: \(e^S_j\).update2.add\((e^S_i)\)
25: \(e^S_i\).update1.add\((e^S_j)\)
26: end if
27: end for
28: if \(e^S_i\).dominationCount < \(k\) then
29: \{ only return relevant entries that can not be pruned, yet \}
30: l.add\((e^S_i)\)
31: end if
32: end for
33: return l

In each call of `joinEntry()`, a lower bound of the number of objects dominating \(e^R\) with respect to \(e^S_i\) is updated for each entry \(e^S_i \in Q^S\). This lower bound is denoted as domination count. Clearly, if for any entry \(e^S_i\), it holds that the domination count ≥ \(k\), then the pair \(< e^R, e^S_i >\) can be safely pruned. Note that using the notion of domination count, the list \(e^S_i\).update1 can be interpreted as the list of entries \(e^S_j\), for which the domination count of \(e^S_j\) may be increased by refinement of \(e^S_i\). The list \(e^S_i\).update2 can be interpreted as the list of entries whose refinement may increase the domination count of \(e^S_i\). In Line 4
of Algorithm 1, the domination count of $e_i^S$ is updated by calling $Dom(e_j^S, e_R, e_i^S)$ for each entry $e_j^S$ in the list $e_i^S.update2$. If $Dom(e_j^S, e_R, e_i^S)$ holds, then the domination count of $e_i^S$ is increased by the number of objects in $e_j^S$. The number of leaf entries is stored in each intermediate entry of the index. Otherwise, i.e., if $e_j^S$ does not dominate $e_R$ w.r.t. $e_i^S$, we check if it is still possible that any point in $e_j^S$ dominates points in $e_R$ with respect to any point in $e_i^S$. If that is not the case, then $e_j^S$ is removed from the list of $e_i^S.update2$, and $e_i^S$ is removed from the list of entries $e_j^S.update1$ (Lines 9-10). If these checks have increased the domination count of $e_i^S$ to $k$ or more, we can safely prune $e_i^S$ in Line 15 and remove all its references from the $update1$ lists of other entries; this is achieved by the delete function.

Now that we have updated domination count values of all $e_i^S \in Q^S$, we start our refinement round in Line 20. Here, we have to decide which entry to refine. We can refine the outer entry $e^R$, or we can refine some, or all entries in the queue of inner entries $Q^S$. A heuristic that has shown good results in practice, is to try to keep, at each stage of the algorithm, both inner and outer entries at about the same volume. Using this heuristic, we first refine all inner entries $e_i^S \in Q^S$ which have a larger volume than the outer entry $e^R$ in line 24. The corresponding algorithm is introduced in the next section.

After refining entries $e_i^S$, we next check in Line 25 if both inner entry $e_i^S$ and outer entry $e^R$ currently considered are both point entries. If that is the case, clearly, neither entry can be further refined, and we perform a $kNN$ query using $e_i^S$ as query object to decide whether $e^R$ is a $kNN$ of $e_i^S$, and, if so, return the pair $e^R, e_i^S$ as a result. Finally, all entries $e_i^S$ which could neither be pruned nor returned as a result, are stored in a new queue $Q_C^S$. This queue is then used to refine the outer entry $e^R$: For each child of $e^R$, the algorithm $joinEntry$ is called recursively, using $Q_C^S$ as inner queue.

### 3.3 Refinement: The resolve-Routine

Our algorithm for refinement of an inner entry $e^S$ is shown in Algorithm 2 and works as follows: We first consider the set of entries $e_i^S.update1$ of other inner entries $e_j^S$ whose domination count may be increased by children $e_i^S$ of $e^S$. For each of these entries, we first remove $e^S$ from its list $e_j^S.update2$, since $e^S$ will be replaced by its children later on. Although $e^S$ does not dominate $e^R$ w.r.t. $e_j^S$, the children of $e^S$ may do. Thus, for each child $e_i^S$ of $e^S$, we now test if $e_i^S$ dominates $e^R$ w.r.t. $e_j^S$ in Line 6 of Algorithm 2. If this is the case, then the domination count of $e_j^S$ is incremented according to the number of objects in $e_i^S$. Otherwise, we check if it is possible for $e_i^S$ to dominate $e^R$ w.r.t. $e_j^S$, and, if that is the case, then $e_j^S$ is added to the list $e_i^S.update1$ of entries which $e_i^S$ may affect, and $e_i^S$ is added to the list $e_j^S.update2$ of entries which may affect $e_i^S$. Now that we have checked which objects the children $e_i^S$ of $e^S$ may affect, we next check which other entries may affect a child $e_i^S$. Thus, we check the list $e_i^S.update2$ of entries which may affect the domination count of $e^S$. For each such entry $e_i^S$, and for each child $e_i^S$, we check

---

1 The check, whether the new domination count of $e_j^S$ exceeds $k$ will be performed in Line 21 of Algorithm 1.
if $e_j^S$ dominates $e^R$ w.r.t. $e_i^S$. If that is the case, the domination count of $e_i^S$ is adjusted accordingly. Otherwise, if $e_j^S$ can possibly dominate $e^R$ w.r.t. $e_i^S$, then we add $e_j^S$ to the list of entries $e_i^S$.update2, and we add $e_j^S$ to the list $e_i^S$.update1. Finally, all child entries of $e^S$ are returned, except those child entries, for which their corresponding domination count already reaches $k$.

4 Experiments

We evaluate our mutual pruning approach using update lists (referred to as UL) in comparison to the state-of-the-art single $R_k$NN query processor TPL in an $R_k$NN join setting within the Java-based KDD-framework ELKI[AGK+12] on both synthetic and real data sets. We use the synthetic data to show the behaviour of the different algorithms in a well-defined setting. Additionally, we use the real data set to show the behaviour of the different algorithms on a not normally distributed data set with dense clusters and additional noise.

As performance indicators we chose the CPU time and the number of page accesses. For measuring the number of page accesses, we assumed that a given number of pages fit into a dedicated cache. If a page has to be accessed but is not contained in the page cache, it has to be reloaded. If the cache is already full and a new page has to be loaded, an old page is kicked out in LRU manner. The page cache only manages data pages from secondary storage, remaining data structures have to be stored in main memory.

Concerning the nomenclature of the algorithms we use the following notation. UL is the mutual pruning based algorithm from Section 3. The additional subscript $S$ (Single) means that every single point of $R$ was queried on its own. With UL$_G$ (Group), a whole set of points, a leaf page, was queried at once. UL$_P$ (Parallel) traversed both indexes for $R$ and $S$ in parallel. These three versions can be easily derived from Algorithm 1 in Section 3. The algorithm expects an entry of $R$’s index. In our performance analysis we call the algorithm with leaf entries (leading to UL$_S$), the entries pointing to leaf nodes (leading to UL$_G$) and the root entry of $R$’s index (leading to UL$_P$). This is especially of interest for large data sets, since UL$_G$ and UL$_S$ allow to split the join up to process it on several distributed systems, increasing its applicability for distributed databases.

TPL was implemented as suggested in [TPL04], however we replaced the clipping step and instead implemented the decision criterion from [EKK+10] to enable cheap pruning on intermediate levels of the indexes.

As an index structure for querying we used an aggregated R*-tree (aR*-Tree [PKZT01]). The page size was set to 1024 bytes, the cache size to 32768 bytes.

4.1 Experiments on Synthetic Data

We chose the underlying synthetic data sets from $R$ and $S$, which have been created with the ELKI-internal data generator, to be normally distributed with equivalent mean and a standard deviation of 0.15. We set the default size of $R$ to $|R| = 0.01|S|$, since the performance of both algorithms degenerates with increasing $|R|$. For each of the analyzed
algorithms we used exactly the same data set given a specific set of input variables in order to reduce skewed results.

During performance analysis, we analyzed the impact of $k$, the number of data points in $R$ and $S$, the dimensionality $d$, and the mean difference $\Delta \mu$ between the data sets $R$ and $S$ on the performance of the evaluated algorithms keeping all but one variable at a fixed default value while varying a single independent variable. Input values for each of the analyzed independent variables can be found in Table 1. In the table, bold values denote default values that are used whenever a different variable is evaluated.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>5, 10, 100, 500</td>
<td>points</td>
</tr>
<tr>
<td>$</td>
<td>R</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>S</td>
<td>$</td>
</tr>
<tr>
<td>$\Delta \mu$</td>
<td>0.0, 0.2, 0.4</td>
<td>$</td>
</tr>
<tr>
<td>$d$</td>
<td>2, 3, 4</td>
<td>dimensions</td>
</tr>
</tbody>
</table>

Table 1: Values for the evaluated independent variables. Default values are denoted in bold.

**Varying $k$.** In a first series of experiments, we varied the parameter $k$. Note that both mutual pruning approaches, TPL and our UL approach are mainly applicable to low values for $k$, especially concerning the execution time (cf. Figure 3 (a)). The runtime of TPL increases considerably fast. The reason for this is that not only the number of result candidates but also the number of objects which are necessary in order to confirm (or prune) these candidates increase superlinear in $k$. In contrast, the runtime of the UL algorithms degenerates slower compared to TPL. The main problem with this family of algorithms is their use of update lists. Each time a page is resolved, the corresponding update lists have to be partially recomputed. This leads to an increase of cost with larger $k$ since on the one hand side, more pages have to be resolved, and on the other hand the length of the update lists of an entry increases and therefore more distance calculations are necessary. Note that ULG and ULP perform very similar to ULs, which is an interesting observation, since for $kNN$ joins parallel tree traversals usually show a higher gain in performance than in an $RkNN$ setting. Concerning the number of page accesses, the picture is quite similar (cf. Figure 4 (a)). TPL shows a performance worse than UL.

**Varying the Size of $R$ ($|R|$).** Varying $|R|$ shows a negative effect on both approaches TPL and UL — their computational time increases considerably fast (cf. Figure 3 (b)). For ULs and TPL the increase of CPU time is linear since these algorithms perform a single $RkNN$ query for each point in $R$. For larger $|R|$, the remaining approaches $UL_G$ and $UL_P$ show a better performance, since these algorithms traverse the tree less often. Interestingly, the number of page access (cf. Figure 4 (b)) for all $UL$ approaches is similar, but always better than for TPL. We explain the large difference in page accesses by the different pruning approaches used by TPL and UL. TPL only employs candidate points for pruning pages, while the UL approaches can also take not yet resolved pages to prune. This can lead to a significant reduction in the number of page accesses.
Varying the Size of S (|S|). Next we analyzed the effect of different values for |S| regarding the CPU time (cf. Figure 3 (c)). Again, the UL approaches perform best, more precisely $UL_S$ since this approach enables highest pruning power. Taking a look at the number of disk accesses (cf. Figure 4 (c)), the results look very similar, however the higher pruning power of $UL_S$ does not show any effect here.

Varying the Overlap Between $R$ and $S$ ($\Delta \mu$). Until now we assumed that the normally distributed sets of values $R$ and $S$ overlap completely, i.e. both sets have the same mean. This assumption is quite intuitive for example if we assume that $R$ and $S$ are drawn from the same distribution. However, if for example $R$ contains feature vectors of a set of dog pictures and $S$ describes mostly flowers, the feature vectors from $R$ and $S$ should be
located at different positions in feature space. We model this behaviour by decreasing the overlap of the two sets $R$ and $S$ and therefore increasing their mean difference ($\Delta\mu = \mu_R - \mu_S$).

Both approaches, UL and TPL can take quite some profit from lower overlap between the sets $R$ and $S$. All of them employ pruning to avoid descending into subtrees that do not have to be taken into account to answer the query. If the overlap decreases, subtrees can be pruned earlier (because the MINDIST between a subtree and the query point increases), greatly reducing the CPU-time and number of page accesses (cf Figures 3 (d) and 4 (d)). Note that for TPL this gain is slightly higher, however even for a mean difference of 0.4, the UL approaches perform better than TPL.
Varying the Dimensionality \((d)\). Taking a look at the performance of the different algorithms with varying dimensionality offers other interesting results (cf. Figure 3 (e) and 4 (e)).

With a dimensionality of 2 and 3, the most important ones for spatial query processing, the UL approaches perform better than TPL concerning the execution time of the algorithms. For two dimensions the gain in performance reaches a factor of 8, for three dimensions still a factor of about 2.6. Beginning with a dimensionality of 4, the UL approaches scale worse than the other approaches concerning execution time, because the pruning power of index-level pruning decreases with increasing dimensionality. With increasing \(d\), the number of entries in an update list increases exponentially. Therefore, much more entries have to be checked each time an intermediate node is resolved, leading to a significant drop in performance.

The results in terms of the number of disk accesses look very similar, therefore they shall not be further investigated. However note that the UL approaches show much better performance in terms of the number of disk accesses than TPL, since they employ pruning on an index level.

4.2 Experiments on Real Data: Postoffice Data Set

Now let us take a look at experiments driven with real data. As a real data set we employed a set of 123593 post offices in the north-eastern united states.\(^2\) The set is clustered (and therefore correlated) in the metropolitan areas, containing further noise in the rural areas, as it can be seen in the visualization of the data set in Figure 5, containing 5000 sample points. Both sets \(R\) and \(S\) are taken from the data set by assigning each of the 123593 points to one of the sets \(R\) or \(S\), respectively. To take full advantage of the whole data set size of 123593 points, we decided to vary the sizes of \(R\) and \(S\) simultaneously such

\(^2\)www.rtreeportal.org/
that $|R| + |S| = 123593$. Clearly, the UL algorithms outperform TPL on this data set (cf. Figure 6). Note that both approaches, TPL and UL, perform better if $R$ is small and $S$ is large than if $S$ is small and $R$ is large. The explanation for this behaviour becomes most clear when taking a look at TPL: Here, the size of $S$ has a lower influence on the performance of the algorithm, because often a larger set $S$ just allows pruning more points. In contrast, increasing the size of $R$ introduces more R$k$NN queries, which is expensive. This problem however, can be mitigated by using $UL_P$ or $UL_G$, since these approaches perform index-level pruning with whole sets of points from $R$.

5 Conclusions

In this paper, we addressed the problem of running multiple R$k$NN-queries at a time, a.k.a R$k$NN join. For this purpose, we proposed a dedicated algorithm for R$k$NN join queries based on the well-known mutual pruning paradigm and evaluated it in a variety of settings including synthetic and real data sets.

However, our research is still preliminary and there is great space for improvements. For example, we would like to develop algorithms specialized for higher dimensionality, since all evaluated algorithms significantly drop in performance for a high number of dimensions. To achieve this, we would like to develop algorithms based on the self pruning paradigm and compare these to the developed mutual pruning approaches.

Acknowledgements. Part of this work was supported by the Deutsche Forschungsgemeinschaft (DFG) under grant number KR 3358/4-1.
References


