Reverse Nearest Neighbors Search in Ad-hoc Subspaces*

Man Lung Yiu
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
mlyiu2@cs.hku.hk

Nikos Mamoulis
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
nikos@cs.hku.hk

Abstract

Given an object q, modeled by a multidimensional point, a reverse nearest neighbors (RNN) query returns the set of objects in the database that have q as their nearest neighbor. In this paper, we study an interesting generalization of the RNN query, where not all dimensions are considered, but only an ad-hoc subset thereof. The rationale is that (i) the dimensionality might be too high for the result of a regular RNN query to be useful, (ii) missing values may implicitly define a meaningful subspace for RNN retrieval, and (iii) analysts may be interested in the query results only for a set of (ad-hoc) problem dimensions (i.e., object attributes). We consider a suitable storage scheme and develop appropriate algorithms for projected RNN queries, without relying on multidimensional indexes. Our methods are experimentally evaluated with real and synthetic data.

1 Introduction

Consider a set $\mathcal{D}$ of objects that are modeled as points in a multidimensional space, defined by the domains of their various features. Given a query object q, a *reverse nearest neighbor* (RNN) query [14, 19, 23, 20, 5, 15, 18, 22, 24] retrieves all objects in $\mathcal{D}$ that have q closer to them than any other object in $\mathcal{D}$ (according to a distance measure). RNN queries are used in a wide range of applications such as decision support, resource allocation, and profile-based marketing.

Assume, for example, that $\mathcal{D}$ is a set of films in a database (owned by a video rental shop) and that each dimension is the rating of the film based on its relevance to a different category (e.g., action, comedy, detective, horror, political, historical, etc.). The rating of a film at a particular dimension is determined by averaging the opinions of customers who have watched the film. Figure 1 shows a few films as points in a multidimensional space, considering only two dimensions; action and comedy. In this space, a and e are the reverse nearest neighbors of q (based on Euclidean distance); these two points have q as their nearest neighbor (NN). The query result could be used to recommend q to customers who have watched a or e, since they could be interested in q, as well. Note that the NN of q (i.e., b) is not necessarily the RNN of q (since c is closer to b), thus NN and RNN queries are essentially two different problems. In addition, RNN queries can have an arbitrary number of results, as opposed to NN queries which have exactly one result.

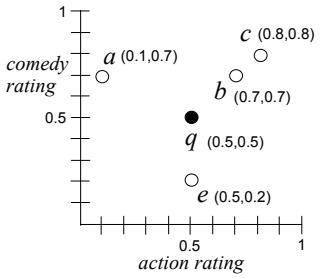


Figure 1. Films rating database

We have illustrated RNN queries based on only two dimensions, however, there may be a large number of dimensions, in general. According to [6], NN search (and RNN search, by extension) could be meaningless in high dimensional spaces, due to the well-known curse of dimensionality. This fact motivated database researchers to study range queries [17], clustering [3], and similarity search [12] in dimensional subspaces where they could be meaningful. The searched subspace is ad-hoc and may vary between different similarity (and RNN) queries. For instance, assume that a new film is registered in the database and watched by some customer. The customer rates the film only based on three dimensions (e.g., action, detective, political), while leaving the rest of the ratings blank. In this case, there is no other meaningful way to search for the RNN of the film, but using only these three dimensions. [10, 17] stress the need for queries in attribute subspaces, due to the existence of missing values. Such *projected* RNN queries could also be applied if some attributes of the query tuple are not relevant

*Work supported by grant HKU 7380/02E from Hong Kong RGC.

for search [10]. A data analyst could explicitly select an ad-hoc dimensional subspace to search, which he thinks interesting. This scenario is very common in business analysis tasks, which aggregate data based on ad-hoc dimensional subspaces in on-line analytical processing (OLAP) applications. Thus, we argue that projected NN and RNN queries in ad-hoc dimensional subspaces are as important as their counterparts that consider the full dimensional space, especially in very high dimensional data collections.

Surprisingly, in spite of the huge bibliography in OLAP [1], to our knowledge, there is no prior work on NN and RNN search in ad-hoc dimensional subspaces. Regarding NN queries, we can attribute this lack of research to the fact that they can be straightforwardly converted to (and solved as) top-k queries [9], as we will discuss later (Section 3.2). However, RNN retrieval is more complex and there is no straightforward adaptation of existing work [14, 19, 23, 22] for the projected version of this problem.

In this paper, we fill this gap by proposing appropriate projected RNN evaluation techniques. Our solution is based on the *decomposition storage model* (DSM) [8]. A commercial product [2] and a prototype DBMS [7, 21] have been developed based on DSM. In DSM, a binary table is created for each attribute, storing for each original tuple, the ID of the tuple and the value of the attribute in that tuple. The binary table can be sorted on attribute value and/or could be indexed by a B^+–tree to facilitate search. As a result, only relevant tables need to be accessed for a query that relates to an ad-hoc set of attributes/dimensions. Vertically fragmented data can be both centralized and distributed. In the distributed model, the binary tables are located at separate servers and remotely accessed by users operating client machines. The queried data are transferred in a form of a stream that stops when the whole result is transmitted or the server receives a termination message from user. A popular example of querying decomposed distributed data is combining object rankings from different sources [9].

The main objective of our DSM-based RNN algorithms is to minimize the number of accesses at the binary tables, since they reflect I/O cost in the centralized model and communication cost in the distributed model. The minor objectives are to reduce computational time and memory usage. The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 defines the problem and outlines the RNN algorithmic framework. Sections 4 and 5 present our methodology. Section 6 discusses evaluation of RkNN queries. Experimental results are presented in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

Section 2.1 surveys RNN algorithms. We review top-k queries in Section 2.2. Section 2.3 discusses related work on dynamic computation of Voronoi cells.

2.1 Euclidean RNN Search

Early RNN algorithms [14, 23] pre-compute the NN distance for each data point p. An index is built on the points with their NN distances so that RNN results can be retrieved fast. This approach has expensive update cost for dynamic datasets. In addition, since we want to support RNN search in ad-hoc sets of dimensions, it is infeasible to materialize the RNN for all points, in all possible subspaces.

Recent methods do not rely on pre-computation, following a filter-refinement framework. In the *filter* step, a set of candidate RNN points (i.e., a superset of the actual result) are retrieved. During the *refinement* (verification) step, a range query is applied around each candidate p to verify whether the query point q is closer to p than any other point in the database. If so, p is reported as an RNN of q. The algorithms of [19, 22] apply on R–trees and rely on geometric properties of the Euclidean distance. [19] divides the (2D) data space around q into 6 regions, such that the RNN of q in each region is exactly the NN of q, there. Thus, in the filter step, 6 *constrained* NN queries are issued to find the candidates. [22] proposes a more efficient geometric solution (TPL) for the filter step that extends the incremental NN (INN) algorithm of [13]. The original INN algorithm [13] first inserts all root entries of the R–tree into a priority queue based on their distance from q. The nearest entry to q is retrieved from the queue; if it is a leaf entry, the corresponding object is reported as the next nearest neighbor. If it is a directory entry, the corresponding node is visited and its entries are inserted into the queue. TPL when visiting a node, before inserting its entries into the priority queue, *trims* their *minimum bounding rectangles* (MBRs) using the already computed RNN candidates to smaller rectangles, by pruning the areas of them which may not contain RNN results. Figure 2 shows an example after two candidate points a and b are discovered. Assume that point a is retrieved first and let M be the MBR of a node not accessed yet. The perpendicular bisector $\perp(a, q)$ of points a and q partitions the data space into two regions: halfplane $\perp_q(a, q)$ containing points closer to q than a, and halfplane $\perp_a(a, q)$ containing points closer to a than q. Note that $\perp_a(a, q)$ cannot contain any RNN results, thus, we only need to consider $M \cap \perp_q(a, q)$ in subsequent search.

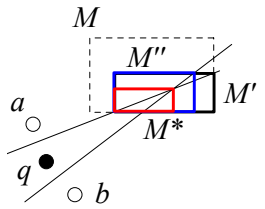


Figure 2. Example of TPL

Since the exact $M \cap \perp_q(a, q)$ may have complex representation (the MBR could be trimmed by multiple bisectors and could be of high dimensionality), [22] suggested to

approximate the trimmed region by its MBR. Consecutive clipping of an MBR is applied if there are multiple candidate RNNs intersecting it. For instance, M is clipped first to M' and then to M'', after considering $\perp(a, q)$ and $\perp(b, q)$ in this order. Although this clipping technique has low computational cost, it may not result in the smallest possible MBR. Observe that the best MBR enclosing the unpruned region in M is M^* instead of M''.

RNN search is a popular problem, many variants of which have been proposed. [18] proposes an approximate algorithm, which cannot guarantee the discovery of all results. [20] focus on *bichromatic* RNN queries. [5] investigate RNN queries on spatiotemporal data. [15] examine aggregate RNN queries, which return an aggregate of the RNN set, on 1D data streams. Finally, [24] study RNN queries on graphs where the distance between two objects is determined by their shortest path. The main defect of existing RNN methods is that they rely either on materialization of results or on multi-dimensional indexes (e.g., R–trees), thus they are not effective in solving the *projected* RNN problem stated in the Introduction. The dataset may have a large number of dimensions and the user could select only an arbitrary, small, *interesting* subset of them, which is different from query to query. Construction and maintenance of numerous (i.e., $2^d - 1$ for d dimensions) specialized indexes for all attribute subsets is too expensive (or infeasible for vertically fragmented distributed data). Besides, existing techniques [19, 22] rely on geometric properties specific for the Euclidean distance, and they cannot be applied for other distance measures (e.g. Manhattan distance).

2.2 Top-k Queries

Our problem is closely related to top-k queries. Given a set of objects and a number of rankings for these objects according to different criteria, a top-k query retrieves the k objects with the highest combined score. Assume for example that we wish to retrieve the restaurants in a city in decreasing order of their aggregate scores with respect to how cheap they are, their quality, and their closeness to our hotel. If three separate services can incrementally provide ranked lists of the restaurants based on their scores in each of the query components, the problem is to identify the k restaurants with the best combined (e.g., average) score.

There are two types of primitive operations used by top-k algorithms: random accesses and sorted accesses. A random access retrieves the value of a particular object (given its ID) for a particular dimension (i.e., attribute). The alternative (sorted accesses) is to retrieve objects from each ranking sequentially, in decreasing order of their scores. The two main top-k retrieval paradigms [9] are: the Threshold Algorithm (TA), which applies both sequential and random accesses and No Random Accesses (NRA), which applies only sorted accesses. They share the following common points. Objects are retrieved from different sources by sorted accesses. A threshold T is defined as the sum of the latest values seen by sorted accesses in all dimensions. Search terminates when the k-th best score is higher than T, in the worst case. Whenever TA sees an object by a sorted access, the values of the object in other dimensions are retrieved by using random accesses and its overall score is computed. On the other hand, NRA applies sorted accesses only. Objects which have been seen in some ranking list are organized based on their overall score in the worst case (assuming minimal values for dimensions where the object has not been seen). TA usually requires fewer accessed in finding the top-k result, however, at the expense of possibly expensive random accesses that are avoided by NRA. In Section 3.2 we elaborate on the relationship between top-k queries and projected NN (and RNN) search.

2.3 Computation of Voronoi Cells

The Voronoi diagram [16] of a dataset $\mathcal{D}$ partitions the space into a number of cells (polygons), one for each point in $\mathcal{D}$, such that for every $p \in \mathcal{D}$, every point inside the Voronoi cell $V(p)$ (of p) is closer to p than any other point in $\mathcal{D}$. We now briefly discuss how Voronoi cells can be used for RNN search. Let q be the query point and P^* be the set of points whose Voronoi cell shares common border with that of $V(q)$. It turns out that the RNN set of q is a subset of P^*. The reason is that every point p',$p' \neq q, p' \notin P^*$ is nearer to some point in P^* than to q.

Computation and maintenance of the Voronoi diagram for every combination of dimensions, for any distance measure, and for any k (for reverse k nearest neighbor retrieval) is infeasible. As a result, a useful operation for RNN search is the computation of $V(q)$ in an ad-hoc dimensional subspace and for any distance measure. In the literature, most Voronoi cell computation methods are appropriate for the Euclidean distance metric only. [20] propose a method for computing an approximation (superset) of $V(q)$, by using a set of NNs around q. [25] suggests a technique for computing the exact Voronoi cell of q. Both methods are based on intersections of bisectors, which is computationally expensive. In addition, they are appropriate for 2D data; in Section 4.2, we discuss why these methods are expensive in arbitrary dimensionality.

Exact Voronoi cell computation for arbitrary dimensional data is discussed in [16]. The proposed algorithm requires examining all the data points and is very expensive. In this paper, we aim at the computation of an *approximate* Voronoi cell $V_W(q)$ of q, which is based on only a subset $W \subseteq \mathcal{D}$ of points that are currently known. It can be easily shown that $V_W(q)$ (spatially) covers the exact $V(q)$ for any $W \subseteq \mathcal{D}$. [20] discusses a heuristic solution for computing an approximation of $V(q)$ in 2D space only and did not provide any analysis about the approximation quality and

the space complexity of their approximation. [4] proposes an off-line method for computing an approximation of $V(q)$ with asymptotic bounds on approximation quality and space complexity. Such a method requires examining many points in the dataset and it cannot be adapted to solve our problem where the points are discovered on-line. In Section 4.2, we present techniques for approximating Voronoi cells.

3 A Framework for RNN Search

We set up the problem studied in this paper by proposing a storage model based on DSM and a framework for projected NN and RNN queries on this model.

3.1 Problem Setting

We consider a set $\mathcal{D}$ of d-dimensional points. $\mathcal{D}$ is stored in $2d$ binary tables, two for each dimension. The tables A_i^+ and A_i^- correspond to dimension i and have identical contents; the IDs of all points in $\mathcal{D}$ and their values in the i-th dimension. The only difference is that A_i^+ is sorted in ascending order of the values of the i-th attribute, whereas in A_i^- tuples are sorted in the reverse order. We store the same information in two tables in order to be able to access attribute values *sequentially* in both directions (ascending and descending order).[1] Let p_i be the value of the point p in dimension i. Given a value q_i, for all points p satisfying $p_i \geq q_i$ ($p_i < q_i$), their values in the i-th dimension can be retrieved in ascending (descending) order, by searching A_i^+ (A_i^-) for q_i and accessing the remainder of the table *sequentially*. Search can be facilitated by sparse B$^+$–trees, built on top of the binary tables.

We emphasize that only the set of *query* dimensions (instead of all dimensions) are considered during query processing. In the rest of the paper, we use d to denote the number of query dimensions (not the data dimensionality). Our goal is to solve RNN queries based on the above data model. Definition 1 states the result set of a RNN query. Unless otherwise stated, we consider Euclidean distance as the dissimilarity function $dist()$. We shall discuss other distance functions in Section 4.2.

Definition 1 *Given a query point q and a dataset $\mathcal{D}$, a RNN query retrieves the set $RNN(q) = \{p \in \mathcal{D} | dist(p,q) < NNdist(p,\mathcal{D})\}$ where $NNdist(p,\mathcal{D})$ denotes the NN distance of p in $\mathcal{D}$.*

3.2 Incremental Nearest Neighbor Search

In this section we show how to adapt the NRA top-k algorithm [9] for incremental retrieval of projected NN from our storage scheme. The proposed projected NN algorithm is extended to solve projected RNN queries in Section 3.3.

[1] Modern hard disks have huge capacity, so the double storage is not a problem. In addition, for applications with vertically fragmented data over distributed servers, we need not store the data twice. Each server is only required to return two streams of values for A_i^+ and A_i^-.

For each dimension i, tuples greater (smaller) than q_i are retrieved from table A_i^+ (A_i^-), sequentially. The value p_i for a particular point p is either in A_i^+ or in A_i^-. Points which have been seen in some (but not all) dimensions are indexed in memory using a hash table. Let $\Lambda(p)$ be the set of dimensions where point p has been seen. Considering Euclidean distance, we can compute the minimum possible distance of p from q as follows:

$$mindist(q,p) = \sqrt{\sum_{i \in \Lambda(p)} |p_i - q_i|^2 + \sum_{i \notin \Lambda(p)} (\min\{v(A_i^+) - q_i, q_i - v(A_i^-)\})^2} \quad (1)$$

, since, in the best case, p_i is equal to the closest value to q_i seen in either A_i^+ or A_i^- in all dimension where p_i has not been seen yet.[2]

Points which have been seen in all dimensions are removed from the hash table and inserted into a min-heap. Let p_{top} be the top object in this heap. If $dist(q, p_{top})$ is smaller than $mindist(q,p)$ for all other points (including completely unseen points) $p \neq p_{top}$, then p_{top} is output as the next NN. In this way, all NNs are (incrementally) output, or the user may opt to terminate search after a satisfactory set of NN has been output.

3.3 A Framework for RNN Search

As discussed in Section 2.1, RNN algorithms operate in two steps; (i) the *filter step* retrieves a candidate set which contains all the actual results, and (ii) the *verification step* eliminates false hits and reports the actual RNNs. This framework allows us to consider filter algorithms and verification algorithms independently. In this section, we focus on the filter step, because it dominates the overall cost (as verified in our experiments). Verification algorithms will be discussed in detail in Section 5.

Figure 3 shows a high-level pseudocode, describing the framework of RNN algorithms that operate on decomposed data. In simple words, the RNN algorithms expand the space around q, discovering RNN candidates and at the same time constraining the additional space that needs to be searched by exploiting the locations of discovered points. S denotes the MBR of the space that potentially contains RNNs of the query point q, not found yet. Initially, it is set to MBR of the universe U, since there is no information about the location of RNNs before search.

Let $v(A_i^+)$ and $v(A_i^-)$ be last values seen on files A_i^+ and A_i^-, respectively, by sorted accesses at the binary tables. The *accessed space* $A = ([v(A_1^-), v(A_1^+)], [v(A_2^-), v(A_2^+)], \cdots, [v(A_d^-), v(A_d^+)])$, is defined by the minimum bounding rectangle (MBR) of the values seen at all binary tables. First, we set A to the MBR of q. Let C be the candidate set and F be the set of points (false hits) that have been seen in all dimensions, but

[2] If for some dimension i, A_i^+ is exhausted then term $v(A_i^+) - q_i$ is removed. Similarly, if A_i^- is exhausted, term $q_i - v(A_i^-)$ is removed.

are not RNNs. Pruned points are maintained in F in order to assist early identification of whether some candidates are false hits (see line 6 of the algorithm). Initially, both C and F are set to empty. We will illustrate the semantics of C and F shortly.

```
Algorithm Filter(Point q, Sources A)
1.  S:=U; A:=q;
2.  C:=∅; F:=∅;
3.  while (S ⊈ A)
4.    p:=GetNext(A);
5.    Reduce(S, p);
6.    if (∃p' ∈ C ∪ F, dist(p, p') ≤ dist(p, q))
7.      F:=F ∪ {p};
8.    else
9.      C:=C ∪ {p};
10. return C;
```

Figure 3. The Filter Algorithm

The filter algorithm has two core operations; $GetNext$ and $Reduce$. Here, we only state their specifications. Their concrete implementations will be studied in Section 4. The function $GetNext(A)$ probes the set of binary tables A (e.g., in a round-robin fashion) and then returns a *complete point* p whose values in all dimensions have been seen. The function $Reduce(S, p)$ uses p to reduce the search space S.

By Definition 1, if a point p is nearer to some other point p' than q, then p cannot be a RNN of q. In this case, p is said to be *pruned* by p'. At Line 6 of the algorithm, we check whether p can be pruned by some other points in C or F. If so, p is pruned and then added to F. Otherwise, p is added to the candidate set C because it is a potential result. The filter step terminates, as soon as the space to be searched S is completely covered by the accessed space A (i.e., no more candidates can be discovered). Note that if S is covered by A in some dimensions and directions, the corresponding tables are pruned from search. Formally, for each dimension i, let $[S_i^-, S_i^+]$ be the projection of S in i. If $v(A_i^-) < S_i^-$, then stream A_i^- is pruned. Similarly, if $v(A_i^+) > S_i^+$, then stream A_i^+ is pruned.

4 Filter Algorithms

In this section, we propose filter algorithms for RNN search. Section 4.1 discusses an adaptation of the TPL algorithm [22] on our data model. Section 4.2 proposes a carefully designed and efficient RNN algorithm. The algorithms follow the framework of Figure 3, thus we confine our discussion on the implementation of $GetNext$ and $Reduce$ operations.

4.1 The TPL Filter

The TPL filter algorithm adapts the access pattern and pruning techniques of the TPL algorithm [22], however, without relying on R–trees. The $GetNext$ function of TPL returns the next NN of q, by applying the incremental algorithm described in Section 3.2. The $Reduce$ function shrinks the search space S by applying the clipping method of [22] directly on S. Let p be the next NN of q. Formally, $Reduce(S, p)$ returns the MBR enclosing $S \cap \perp_q(p, q)$.

The main disadvantage of the TPL filter is that MBR clipping introduces more dead space than necessary (as discussed in Section 2.1). Thus, it does not prune the search space effectively, increasing the number of accesses. A minor disadvantage is that it employs incremental NN search. In Section 4.2, we show that we can take advantage of points seen in all dimensions, as soon as they are identified, no matter whether they are the next NN of q or not.

4.2 The Greedy Filter

The Greedy filter algorithm is a carefully designed RNN algorithm on our data model, which does not share the drawbacks of the TPL filter algorithm. The $GetNext$ function of our algorithm is not based on incremental NN search. Instead, we modify the process of Section 3.2 to immediately return a point, as soon as it has been seen in all dimensions. The rationale is that *complete* points seen earlier than the next NN may shrink the search space fast, allowing earlier termination of the filter step.

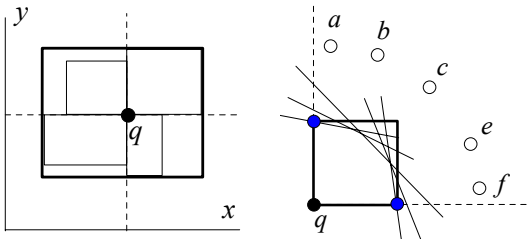
The Greedy filter algorithm also applies an improved method for reducing the search space S. The MBR of the exact Voronoi cell $V(q)$ of q is the minimal search space S, as discussed in Section 2.3. Since $V(p)$ is unknown before search, our algorithm progressively computes a more refined approximation of it while retrieving points. Let W be a set of known (i.e., retrieved) points around q. Based on W, we can compute an approximation $V_W(p)$ of $V(p)$, by taking the intersection of all halfplanes $\bigcap_{p \in W} \perp_q(p, q)$. Halfplane intersection (for L_2 norm) is both computationally expensive and space consuming. According to [16], each incremental computation requires $O(|W|^{\lceil d/2 \rceil})$ time and $O(d|W|^{\lceil d/2 \rceil})$ space (vertices of the resulting Voronoi cell). In addition, computation of halfplanes is far more complex for distance metrics other than L_2. Finally, halfplane intersection cannot be directly applied for RkNN search, which will be discussed in Section 6. We observe that, setting the search space S to any superset of $V_W(q)$ guarantees that no results outside the accessed space A will be missed, thus exact computation of $V_W(q)$ may not be necessary for RNN retrieval. Next, we discuss two methods that compute conservative approximations of $V_W(q)$ that do not rely on halfplane intersection and can be computed for arbitrary L_p distance norms.

4.2.1 Approximation using intercepts

Our first method approximates $V_W(q)$, dynamically and efficiently, as new points are retrieved. In addition, the approximated cell requires only bounded space, which is much smaller than the space required for representing the exact $V(q)$ in the worst case. Initially, we show how this method works with the Euclidean distance and then extend

it for any L_p distance norm.

First, we partition the search space around q into 2^d quadrants, as shown in Figure 4a. Consider the upper right quadrant in this example. Figure 4b illustrates how to derive the (local) search space for this quadrant. Suppose we have discovered 5 points a, b, c, e, f there. For each point p found ($p \in \{a, b, c, e, f\}$), we compute the intercepts of $\perp(p, q)$ with the axes of the quadrant. It turns out that it suffices to compute and maintain the intercept closest to q for each dimension. Let M be the MBR containing q and these intercepts. Lemma 2 (based on Lemma 1) guarantees that M contains all potential RNNs in the quadrant. After M has been computed for all quadrants, the (global) search space S is taken as their MBR, as shown in Figure 4a.



(a) Global search space (b) Local search space

Figure 4. Voronoi cell approximation

Lemma 1 *Consider the quadrant Q with coordinates no smaller than q in all dimensions. Let p be a point in Q and let e be the intercept of $\perp(p, q)$ with some axis r, i.e., $e = (q_1, \ldots, q_{i-1}, c_r, q_{i+1}, \ldots, q_d)$. For any point p', for which $\forall i \in [1, d] : p'_i \geq e_i$, we have $dist(p', q) \geq dist(p', p)$.*

Proof. We first compare $dist(e, q)$ and $dist(e, p)$ with the corresponding distances $dist(p', q)$ and $dist(p', p)$ for every dimension individually. For any dimension i, let $diff_q = |p'_i - q_i| - |e_i - q_i|$ ($diff_q \geq 0$, since $e_i \geq q_i$ and $p'_i \geq e_i$). Similarly, let $diff_p = |p'_i - p_i| - |e_i - p_i|$. If $p'_i \leq p_i$, then $diff_p \leq 0$. If $p'_i \geq p_i$, then $diff_p \leq diff_q$, since $q_i \leq p_i \leq p'_i$. Thus, in any case, $diff_p \leq diff_q$. Since, in *all* dimensions p' can only be closer to p than e is and p' can only be further than q than e is, and due to the monotonicity of the Euclidean distance (based on the atomic dimensional distances), we have $dist(p', q) \geq dist(p', p)$. ∎

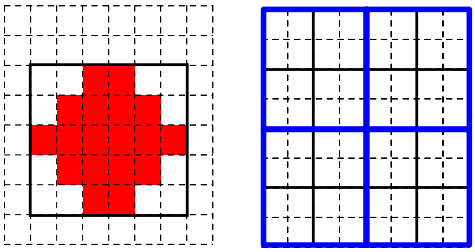
Lemma 2 *Consider a quadrant Q defined by q. Let I be the set of the intercepts that are closest to q for each dimension. Let M be the MBR defined by q and these intercepts. M encloses all RNNs of q in Q that are located outside the accessed space A.*

When multiple points exist in a quadrant, the nearest intercepts to q dominate in pruning. Thus, Lemma 2 can be trivially proved. We can prove versions of Lemmas 1 and 2 for any L_p metric, since the basic proof (of Lemma 1) is based on the monotonicity property of Euclidean distance. An intercept coordinate $e = (q_1, \ldots, q_{i-1}, c_r, q_{i+1}, \ldots, q_d)$ for some axis r, of the halfplane between q and a seen point x, can be easily computed from the equation $dist(e, q) = dist(e, x)$. Thus, our technique can be applied for any L_p norm.

We stress that our Voronoi cell approximation technique is functionally different from the one in [20]. We use intercepts (based on any L_p norm) to compute a rectangle that encloses $V(q)$, whereas [20] compute a more complex 2D approximation of the cell. Thus, our method is applicable for any dimensionality (with significantly lower space requirements) and distance metric. Our approximation method is expected to outperform the TPL filter discussed in Section 4.1, since it optimally clips the quadrants containing points using information about these points. On the other hand, the TPL filter operates on the MBR of the whole search space S, which is harder to prune. The only drawback of our technique is that each retrieved point is not utilized in pruning other quadrants except the one it resides in. In the next section, we propose another pruning technique that utilizes the effect of discovered points in neighboring quadrants.

4.2.2 Approximation using a hierarchical grid

In this section, we propose a method that approximates the MBR that covers $V_W(p)$ with the help of a multi-dimensional grid. This approach has several advantages. First, it provides a guarantee on the quality of the approximation. Second, no memory is needed for storing the cells. Third, this technique can directly be used for other distance metrics. Initially, we assume that the Euclidean distance is used; later we discuss other distance metrics.



(a) a simple grid (b) a hierarchical grid

Figure 5. Reducing search space using a grid

Figure 5a shows an exemplary 8×8 grid that partitions the search space S. Whenever a new point is retrieved by $GetNext$, we check (by the use of bisectors) whether a cell can be pruned by the points which have been seen in all dimensions. If not, the cell (shown in red) is included in the revised search space S' for the next round (to be used for the next retrieved point). Instead of explicitly including all non-pruned cells in S', we consider the MBR of them (since the decomposed tables are essentially accessed until the MBR

of $V(q)$ anyway). Thus, we need not explicitly maintain in memory any grid information. When the algorithm is invoked for the next point, the search space S is smaller than before, thus the cells become smaller and the approximation quality improves incrementally.

Yet, the drawback of the above technique is that it requires high computational cost, especially in high dimensional space, since a large number of cells must be checked. In order to reduce the CPU cost we introduce a hierarchical grid, as shown in Figure 5b and employ branch-and-bound techniques to speed up computation. We first attempt to prune a high level cell. If pruning fails, then we partition it into smaller ones and apply the above procedure recursively. In Figure 5b, the maximum recursion level is set to 3. This parameter is a tradeoff between the approximation quality and the computational cost.

Figure 6 shows this hierarchical grid based traversal algorithm for search space reduction. First, the newly discovered point p is added to the set of points W, used for pruning (i.e., $W = C \cup F$). Then, we dynamically impose a hierarchical grid to the current search space S and prune its cells hierarchically. S' denotes the output search space (MBR of cells that are not pruned). In the $Traverse$ function, a cell e is examined when (i) it is not covered by S', and (ii) it cannot be pruned by any points in W. At recursion level 0, pruning terminates and S' is enlarged to cover e. Note that the output S' of this algorithm is guaranteed to be no larger than $2\overline{e}$ than the exact MBR of $V_W(q)$, in each dimension, where $\overline{e}$ is the length of a cell at the finest grid resolution. As a result, the proposed technique provides a good approximation guarantee.

```
Algorithm Grid-Reduce(MBR S,Point p)
1.  Global Set W; // reuse content in previous run
2.  W:=W ∪ {p};
3.  S':=∅;
4.  Traverse(S', S,MAX_LEVEL, q, W);
5.  S:=S';

Algorithm Traverse(MBR S',Cell e,Int level,Point q,Set W)
1.  if (e ⊈ S') // e not covered by S'
2.    if (∀p ∈ W, e cannot be pruned by p)
3.      if (level = 0)
4.        enlarge S' to cover e;
5.      else
6.        partition e into 2^d sub-cells;
7.        for each cell e' ⊆ e
8.          Traverse(S', e', level − 1, q, W);
```

Figure 6. Hierarchical Grid Traversal

The grid-based Greedy filter algorithm can be applied for other distance metrics by using alternative pruning methods for cells (i.e., not based on perpendicular bisectors), described by Lemma 3 (straightforwardly proven).

Lemma 3 *Let M be a rectangle. For any distance metric, if $maxdist(p, M) \leq mindist(q, M)$ then $\forall p' \in M, dist(p, p') \leq dist(q, p')$.*

5 Verification of Candidates

In this section, we discuss whether the candidates obtained in the filter step are actual RNNs. In addition, we discuss early (progressive) computations of RNNs before the verification step. Finally, we show a method that minimizes F, i.e., the set of points that are not candidates, but they are used to prune C.

5.1 Concurrent Verification

The filter step terminates with a set C of candidate points and a set F of false hits; points that have been seen in all dimensions, but they are found not to be RNNs. Normally, each candidate $p \in C$ is verified by issuing a range search around p with radius $dist(q, p)$. If another point is found within this range then p is not an RNN of q, otherwise it is returned. In order to reduce the number of range queries, we perform verification in two steps. First, we check each $p \in C$ whether they are closer to some other seen point in $C \cup F$ than to q. These candidates can be immediately eliminated. The second step is to check the remaining candidates by range queries. Instead of issuing individual queries for each candidate, we perform a *concurrent verification*, which continues traversing the binary tables from the point where the filter algorithm has stopped, until all candidates have been verified. The overall verification algorithm is shown in Figure 7. The main idea of the second step is to compute a rectangle M for each candidate p (based on $dist(q, p)$), where its potential neighbors closer than q may be contained. While accessing the binary tables in search for these point, each complete point w is checked on whether it can prune any of the remaining candidates in C (not only p). If p cannot be pruned, then it is reported as a result.

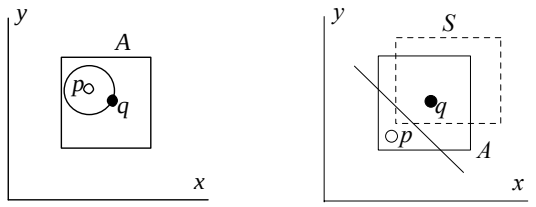
```
Algorithm Concurrent-Verification(Sources A,Candidate Set C,False Hit Set F)
1.  C:=C − {p ∈ C|∃p' ∈ (F ∪ C − {p}), dist(p, p') ≤ dist(p, q)};
2.  for each p ∈ C
3.    δ:=dist(p, q);
4.    M:=([p_1 − δ, p_1 + δ], [p_2 − δ, p_2 + δ], ··· , [p_d − δ, p_d + δ]);
5.    while (p ∈ C ∧ M ⊈ A) // p not removed and M not completely accessed
6.      w:=GetNext(A);
7.      C:=C − {p' ∈ C|dist(p', w) ≤ dist(p', q)};
8.    if (p ∈ C)
9.      report p as a result;
```

Figure 7. Concurrent Verification Algorithm

5.2 Progressive RNN Computation

Our algorithmic framework allows early report of points that are definitely in the RNN set, before the verification phase. Progressive report of results is very useful in practice, since the user can examine early results, while waiting for the complete response set. Given a candidate point p, let $M(p)$ be the MBR enclosing the region with p as center and the range as $dist(p, q)$. Formally, we have $M(p) = ([p_1-\delta, p_1+\delta], [p_2-\delta, p_2+\delta], \cdots, [p_d-\delta, p_d+\delta])$, where $\delta = dist(p, q)$. During the filter step, if a candidate p satisfies (i) $M(p) \subseteq A$, and (ii) $\forall p' \in (C \cup F -$

$\{p\}), dist(p, p') > dist(p, q)$, then p can be immediately reported as a result. In the example of Figure 8a, $M(p)$ is enclosed in A and does not contain any other point but p.



(a) progressive verification (b) reducing the refinement set

Figure 8. Optimizing the filter step

5.3 Reducing the Set of False Hits

During the filter step, we maintain a potentially large set F of points that are false hits, but may be used for candidate pruning. We can reduce this set, by eliminating points that may not be used to prune any candidate. A point $p \in F$ can be discarded if (i) p does not fall in the verification range of any existing candidate, and (ii) $\perp_p(q, p) \cap S \subseteq A$. $\perp_p(q, p)$ is the part of the data space containing points closer to p than q. Only the points in this region can be pruned by p. If its intersection with the search space S is already covered by the accessed space A, then any complete points found later cannot be pruned by the point p. Note that this condition can be generalized for arbitrary distance metrics, by replacing $\perp_p(q, p)$ by the region closer to p than to q. Figure 8b illustrates an example, where a (non-candidate) point p can be pruned from F.

6 RkNN Search

In this section, we discuss how our framework can be adapted for the generalized problem of RkNN search: find all points p such that q belongs to the k-NN set of p. The TPL filter can be generalized for RkNN search, if we select a k-subset $\{\theta_1, \theta_2, \cdots, \theta_k\}$ of the points in $C \cup F$. Let $clip(S, q, \theta_i)$ be the MBR (in S) that may contain some points closer to the query point q than the point θ_i. Let S' be the MBR that encloses $clip(S, q, \theta_i) \forall i \in [1, k]$. Observe that other RNN results cannot be outside S' because all such points are nearer to all $\theta_1, \theta_2, \cdots, \theta_k$ than to q. Therefore, S' becomes the new search space after a new point has been retrieved. Appropriate k-subsets of $C \cup F$ to be used for pruning can be selected using the heuristics of [22].

The Greedy filter can be adapted for RkNN search, by considering the k-th closest intercept for each axis adjacent to each quadrant. Due to space constraints, the proof for the correctness of this approach is omitted. We stress that this technique is deterministic, as opposed to the probabilistic nature of selecting k-subsets in the TPL filter. In addition, it is applicable to any L_p distance norms. The grid-based Greedy filter can also be easily extended for RkNN search; a cell in this case is pruned if it falls outside $\perp_q(q, p)$ for at least k points $p \in C \cup F$.

For the verification step of RkNN search, for each candidate point p, we keep a counter of the points in $C \cup F$, which are closer to p than to q, during the filter step. Every time a new point is accessed, these counters are updated. Eventually, verification is required only for candidates for which the counter is smaller than k. We note that we have also extended our framework successfully for *bichromatic* RNN queries [20]. Details are omitted due to space constraints.

7 Experimental Evaluation

In this section, we evaluate the proposed RNN algorithms using synthetic and real datasets. All algorithms (TPL, G-IA for Greedy with intercept approximation, and G-HG for Greedy with hierarchical grid) were implemented in C++. All experiments were performed on a Pentium IV 2.3GHz PC with 512MB memory. The maximum recursion level of the search space reduction algorithm in G-HG is fixed to 5 (i.e., a grid of 32^d finest cells). For each experimental instance, the query cost is averaged over 100 queries with the same properties. We considered Euclidean distance in all experiments, since TPL is inapplicable for other distance metrics.

7.1 Experimental Settings

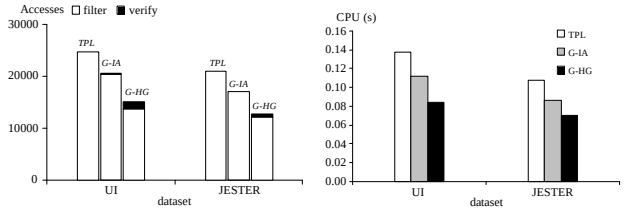
We generated uniform synthetic datasets (UI) by assigning random numbers to attribute values of objects independently. The default number of objects in a synthetic dataset is $N = 100K$. We also used a real dataset (JESTER [11]), which contains a total of 4.1M ratings of 100 jokes from 73K users. A joke may not be rated by all users. We extracted the attributes (i.e., jokes) having value for at least 60K objects (i.e., users) and then constructed binary tables for them (22 attributes). Query objects are users randomly chosen from the dataset. For a particular query object q we use only the attributes for which q has ratings to issue a projected RNN query. In this way, we are able to extract query workloads with a specified number of query dimensions. The query result can be used to recommend q to his/her RNNs as a potential "buddy", since q has similar taste in jokes as them.

Attribute values of both UI and JESTER datasets are normalized to the range $[0, 1]$. We tried different access patterns for sequential accesses to the binary tables during RNN evaluation (i.e., round-robin, equi-depth, etc.). We found no practical difference between these schemes, thus we use a round-robin accessing scheme in all experiments reported here.

7.2 Experimental Results

We study the performance of RNN search with respect to various factors. Figure 9a shows the filter and verification costs (in terms of accesses) of the algorithms on the UI

and JESTER datasets for queries with $d = 3$ dimensions. The filter costs of the algorithms are proportional to their search space. The MBR clipping technique in TPL prunes the space too loosely. G-IA is more effective in space reduction than TPL. Finally, G-HG has the lowest filter cost as it utilizes the pruning power of discovered points in all quadrants. The concurrent verification algorithm is very efficient; verification costs less than 10% of the total cost. Since TPL and G-IA search more space than G-HG, they eventually discover more points than G-HG, which can be used to prune more candidates. This explains the higher verification cost of G-HG compared to the other methods. As Figure 9b shows, the CPU cost of the algorithms follows the same trend as the number of accesses. Unless otherwise stated, we consider JESTER as the default dataset in subsequent experiments.



(a) access cost (b) CPU cost

Figure 9. Cost on different datasets, $d = 3$

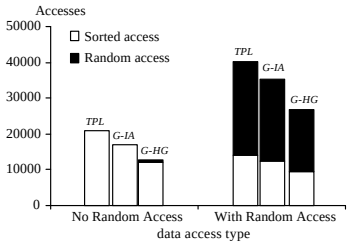
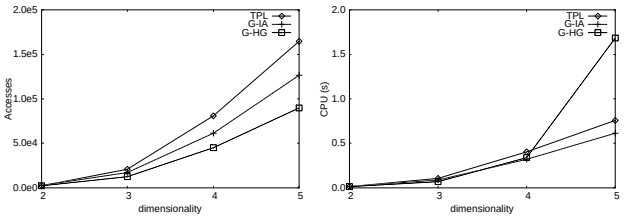


Figure 10. Data access types, $d = 3$, JESTER

The next experiment justifies why we use only sorted accesses to the binary tables, whereas one could develop RNN algorithms that extend TA [9]. We implemented versions of TPL, G-IA, and G-HG that perform random accesses; whenever an object is seen from a binary table, $d - 1$ random accesses to all other tables are applied to retrieve the values of the object in all other dimensions. Thus, there are no *partially* seen objects. Figure 10 compares the original filter algorithms with their versions that employ random accesses (for queries with $d = 3$). Observe that the total access cost when using random accesses is much higher than when not. In practice, their access cost difference is even higher, provided that random accesses are more expensive than sorted ones in real applications.

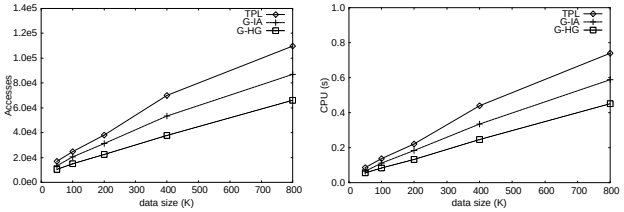
Figure 11 shows the access and CPU cost of the algorithms as a function of query dimensionality d. G-HG outperforms the other algorithms in terms of accesses and the performance gap widens as d increases. The pruning effectiveness of TPL and G-IA decreases with dimensionality. A bisector is less likely to prune all dimensions and reduce the global MBR, thus TPL is not very effective. Besides, for a discovered point p, the number of neighbor quadrants increase with d and G-IA fails to utilize p in pruning them. The CPU cost has a slightly different trend. G-HG becomes very expensive at $d = 5$ (and higher values) because it needs to examine a large number of hierarchical cells. We recommend G-IA for high query dimensionality, because it achieves good balance between accesses and CPU cost.



(a) access cost (b) CPU cost

Figure 11. Cost vs dimensionality d, JESTER

Figure 12 shows the cost of the algorithms as a function of the data size N, on 3D UI datasets. All the algorithms are scalable as their costs increase sub-linearly as N increases. Again, G-HG outperforms the other methods and the performance gap widens as N increases.



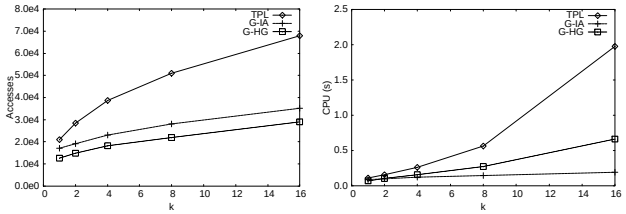
(a) access cost (b) CPU cost

Figure 12. Cost vs data size N, $d = 3$, UI

We also compared the algorithms for RkNN search. Figure 13 shows the performance of the algorithms with respect to k. Access costs of the algorithms increase sub-linearly as k increases. The cost of TPL increases at the fastest rate because it applies a heuristic, which only considers subsets of discovered points in reducing the search space. On the other hand, G-IA and G-HG employ deterministic and systematic approaches for reducing the search space effectively. Regarding CPU cost, TPL is the most expensive as it needs to examine several subsets of points. Also, G-HG becomes more expensive than G-IA at high values of k because some high level (hierarchical) cells cannot be immediately pruned and more low level cells need to be visited.

Figure 14 shows the progressiveness of the algorithms for a typical R4NN query on a 3D UI dataset. All the algorithms generate the first few results early because all of

them follow the same filter framework algorithm. Their effectiveness of reducing the search space only affects their total cost. The arrows indicate that G-HG terminated first, followed by G-IA and TPL. Finally, we compared the performance of G-IA and G-HG for L_1 and L_∞ (TPL is inapplicable in this experiment) and found that G-HG consistently outperforms G-IA in terms of accesses. Detailed results are omitted due to lack of space.



(a) access cost (b) CPU cost

Figure 13. Cost vs k, $d = 3$, JESTER

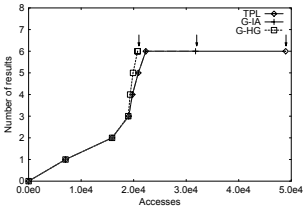


Figure 14. Progressive RNN, $N = 100K$, $d = 3$, $k = 4$, UI

8 Conclusion

We proposed the first algorithms for projected RNN queries (and their variants) on the decomposed data model and evaluated their performance on both synthetic and real datasets. We also proposed the first techniques for retrieving RNN results in a progressive way. The algorithm G-HG requires the least number of data accesses while G-IA is more balanced between access cost and CPU cost. In terms of flexibility, G-HG is applicable to any distance metric, G-IA is applicable to any L_p distance norm, and TPL is only applicable to L_2 norm. In the future, we will study more optimizations for projected RNN algorithms on vertically decomposed data.

References

[1] Data Warehousing and OLAP: A Research-Oriented Bibliography. http://www.ondelette.com/OLAP/dwbib.html.

[2] Sysbase IQ white papers. http://www.sybase.com.

[3] C. C. Aggarwal, C. M. Procopiuc, J. L. Wolf, P. S. Yu, and J. S. Park. Fast Algorithms for Projected Clustering. In *SIGMOD*, 1999.

[4] S. Arya and A. Vigneron. Approximating a Voronoi cell. *HKUST Research Report HKUST-TCSC-2003-10*, 2003.

[5] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *IDEAS*, 2002.

[6] K. Beyer, J. Goldstein, R.Ramakrishnan, and U. Shaft. When is "Nearest Neighbor" Meaningful? In *ICDT*, 1999.

[7] P. Boncz and M. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB Journal*, 8(2):101–119, 1999.

[8] G. Copeland and S. Koshafian. A Decomposition Storage Model. In *SIGMOD*, 1985.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[10] G. H. Gessert. Four Valued Logic for Relational Database Systems. *SIGMOD Record*, 19(1):29–35, 1990.

[11] K. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A Constant Time Collaborative Filtering Algorithm. *Information Retrieval*, 4(2):133–151, 2001.

[12] A. Hinneburg, C. C. Aggarwal, and D. A. Keim. What is the Nearest Neighbor in High Dimensional Spaces? In *VLDB*, 2000.

[13] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *TODS*, 24(2):265–318, 1999.

[14] F. Korn and S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In *SIGMOD*, 2000.

[15] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse Nearest Neighbor Aggregates Over Data Streams. In *VLDB*, 2002.

[16] A. Okabe, B. Boots, K. Sugihara, and S. Chiu. *Spatial Tessellations : Concepts and Applications of Voronoi Diagrams*. Wiley, second edition, 2000.

[17] B. C. Ooi, C. H. Goh, and K.-L. Tan. Fast High-Dimensional Data Search in Incomplete Databases. In *VLDB*, 1998.

[18] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High Dimensional Reverse Nearest Neighbor Queries. In *CIKM*, 2003.

[19] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse Nearest Neighbor Queries for Dynamic Databases. In *SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2000.

[20] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of Influence Sets in Frequently Updated Databases. In *VLDB*, 2001.

[21] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniak, M. Ferreria, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[22] Y. Tao, D. Papadias, and X. Lian. Reverse kNN Search in Arbitrary Dimensionality. In *VLDB*, 2004.

[23] C. Yang and K. I. Lin. An Index Structure for Efficient Reverse Nearest Neighbor Queries. In *ICDE*, 2001.

[24] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse Nearest Neighbors in Large Graphs. In *ICDE*, 2005.

[25] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. Lee. Location-based Spatial Queries. In *SIGMOD*, 2003.