Inverse Queries For Multidimensional Spaces

Thomas Bernecker[1], Tobias Emrich[1], Hans-Peter Kriegel[1], Nikos Mamoulis[2], Matthias Renz[1], Shiming Zhang[2], and Andreas Züfle[1]

[1] Institute for Informatics, Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, Germany
{bernecker,emrich,kriegel,renz,zuefle}@dbs.ifi.lmu.de
[2] Department of Computer Science, University of Hong Kong
Pokfulam Road, Hong Kong
{nikos,smzhang}@cs.hku.hk

Abstract. Traditional spatial queries return, for a given query object q, all database objects that satisfy a given predicate, such as epsilon range and k-nearest neighbors. This paper defines and studies *inverse* spatial queries, which, given a subset of database objects Q and a query predicate, return all objects which, if used as query objects with the predicate, contain Q in their result. We first show a straightforward solution for answering inverse spatial queries for any query predicate. Then, we propose a filter-and-refinement framework that can be used to improve efficiency. We show how to apply this framework on a variety of inverse queries, using appropriate space pruning strategies. In particular, we propose solutions for inverse epsilon range queries, inverse k-nearest neighbor queries, and inverse skyline queries. Our experiments show that our framework is significantly more efficient than naive approaches.

1 Introduction

Recently, a lot of interest has grown for *reverse* queries, which take as input an object o and find the queries which have o in their result set. A characteristic example is the reverse k-NN query [6, 12], whose objective is to find the query objects (from a given data set) that have a given input object in their k-NN set. In such an operation the roles of the query and data objects are reversed; while the k-NN query finds the *data* objects which are the nearest neighbors of a given *query* object, the reverse query finds the objects which, if used as queries, return a given data object in their result. Besides k-NN search, reverse queries have also been studied for other spatial and multidimensional search problems, such as top-k search [13] and dynamic skyline [7]. Reverse queries mainly find application in data analysis tasks; e.g., given a product find the customer searches that have this product in their result. [6] outlines a wide range of such applications (including business impact analysis, referral and recommendation systems, maintenance of document repositories).

In this paper, we generalize the concept of reverse queries. We note that the current definitions take as input a *single* object. However, similarity queries such as k-NN queries and ε-range queries may in general return more than one result. Data analysts are often interested in the queries that include two or more given objects in their result.

Such information can be meaningful in applications where only the result of a query can be (partially) observed, but the actual query object is not known. For example consider an online shop selling a variety of different products stored in a database $\mathcal{D}$. The online shop may be interested in offering a *package* of products $Q \subseteq \mathcal{D}$ for a special price. The problem at hand is to identify customers which are interested in *all* items of the package, in order to direct an advertisement to them. We assume that the preferences of registered customers are known. First, we need to define a predicate indicating whether a user is interested in a product. A customer may be interested in a product if

- the distance between the product's features and the customer's preference is less than a threshold ε;
- the product is contained in the set of his k favorite items, i.e., the k-set of product features closest to the user's preferences;
- the product is contained in the customer's dynamic skyline, i.e., there is no other product that better fits the customer's preferences in every possible way.

Therefore, we want to identify customers r, such that the query on $\mathcal{D}$ with query object r, using one of the query predicates above, contains Q in the result set. More specifically, consider a set $\mathcal{D} \in \mathbb{R}^d$ as a database of n objects and let $d(\cdot)$ denote the Euclidean distance in $\mathbb{R}^d$. Let $\mathcal{P}(q)$ be a query on $\mathcal{D}$ with predicate $\mathcal{P}$ and query object q.

Definition 1. *An inverse $\mathcal{P}$ query (I$\mathcal{P}$Q) computes for a given set of query objects $Q \subseteq \mathcal{D}$ the set of points $r \in \mathbb{R}^d$ for which Q is in the $\mathcal{P}$ query result; formally:*

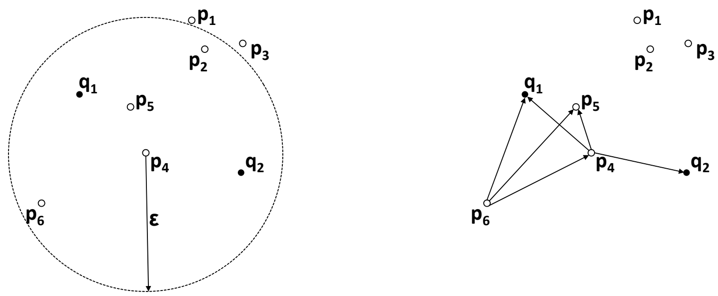
$$I\mathcal{P}Q = \{r \in \mathbb{R}^d : Q \subseteq \mathcal{P}(r))\}$$

Simply speaking, the result of the *general* inverse query is the subset of the space defined by all objects r for which all Q-objects are in $\mathcal{P}(r)$. Special cases of the query are:

- The mono-chromatic inverse $\mathcal{P}$ query, for which the result set is a subset of $\mathcal{D}$.
- The bi-chromatic inverse $\mathcal{P}$ query, for which the result set is a subset of a given database $\mathcal{D}' \subseteq \mathbb{R}^d$.

In this paper, we study the inverse versions of three common query types in spatial and multimedia databases as follows.

Inverse ε-Range Query ($I\varepsilon$-RQ). The inverse ε-range query returns all objects which have a sufficiently low distance to all query objects. For a *bi-chromatic* sample application of this type of query, consider a movie database containing a large number of movie records. Each movie record contains features such as humor, suspense, romance, etc. Users of the database are represented by the same attributes, describing their preferences. We want to create a recommendation system that recommends to users movies that are sufficiently similar to their preferences (i.e., distance less than ε). Now, assume that a group of users, such as a family, want to watch a movie together; a bi-chromatic $I\varepsilon$-RQ will recommend movies which are similar to *all* members of the family. For a mono-chromatic case example, consider the set $Q = \{q_1, q_2\}$ of query objects of Figure 1(a) and the set of database points $\mathcal{D} = \{p_1, p_2, \cdots, p_6\}$. If the range ε is as illustrated in the figure, the result of the $I\varepsilon$-$RQ(Q)$ is $\{p_2, p_4, p_5\}$ (e.g., p_1 is dropped because $d(p_1, q_2) > \varepsilon$).



(a) $I\varepsilon$-RQ.

(b) Ik-NNQ, $k = 3$.

Fig. 1. Examples of inverse queries.

Inverse k-NN Query (Ik-NNQ). The inverse k-NN query returns the objects which have all query points in their k-NN set. For example, *mono-chromatic* inverse k-NN queries can be used to aid crime detection. Assume that a set of households have been robbed in short succession and the robber must be found. Assume that the robber will only rob houses which are in his close vicinity, e.g. within the closest hundred households. Under this assumption, performing an inverse 100NN query, using the set of robbed households as Q, returns the set of possible suspects. A mono-chromatic inverse 3NN query for $Q = \{q_1, q_2\}$ in Figure 1(b) returns $\{p_4\}$. p_6, for example, is dropped, as q_2 is not contained in the list of its 3 nearest neighbors.

Inverse Dynamic Skyline Query (I-DSQ). An inverse dynamic skyline query returns the objects, which have all query objects in their dynamic skyline. A sample application for the *general* inverse dynamic skyline query is a product recommendation problem: assume there is a company, e.g. a photo camera company, that provides its products via an internet portal. The company wants to recommend products to their customers by analyzing the web pages visited by them. The score function used by the customer to rate the attributes of products is unknown. However, the set of products that the customer has clicked on can be seen as samples of products that he or she is interested in, and thus, must be in the customer's dynamic skyline. The inverse dynamic skyline query can be used to narrow the space which the customers preferences are located in. Objects which have all clicked products in their dynamic skyline are likely to be interesting to the customer. In Figure 1, assuming that $Q = \{q_1, q_2\}$ are clicked products, I-$DSQ(Q)$ includes p_6, since both q_1 and q_2 are included in the dynamic skyline of p_6.

For simplicity, we focus on the mono-chromatic cases of the respective query types (i.e., query points and objects are taken from the same data set); however, the proposed techniques can also be applied for the bi-chromatic and the general case. For details, refer to the full version of this paper [2].

Motivation. A naive way to process any inverse spatial query is to compute the corresponding reverse query for each $q_i \in Q$ and then intersect these results. The problem of this method is that running a reverse query for each q_i multiplies the complexity of the reverse query by $|Q|$ both in terms of computational and I/O-cost. Objects that are not shared in two or more reverse queries in Q are unnecessarily retrieved, while objects that are shared by two or more queries are redundantly accessed multiple times.

We propose a filter-refinement framework for inverse queries, which first applies a number of filters using the set of query objects Q to prune effectively objects which may not participate in the result. Afterwards, candidates are pruned by considering other database objects. Finally, during a *refinement* step, the remaining candidates are verified against the inverse query and the results are output. When applying our framework to the three inverse queries under study, filtering and refinement are sometimes integrated in the same algorithm, which performs these steps in an iterative manner. Although for $I\varepsilon\text{-}RQ$ queries the application of our framework is straightforward, for $Ik\text{-}NNQ$ and $I\text{-}DSQ$, we define and exploit special pruning techniques that are novel compared to the approaches used for solving the corresponding reverse queries.

Outline. The rest of the paper is organized as follows. In the next section we review previous work related to inverse query processing. Section 3 describes our framework. In Sections 4-6 we implement it on the three inverse spatial query types; we first briefly introduce the pruning strategies for the single-query-object case and then show how to apply the framework in order to handle the multi-query-object case in an efficient way. Section 7 is an experimental evaluation and Section 8 concludes the paper.

2 Related Work

The problem of supporting reverse queries efficiently, i.e. the case where Q only contains a single database object, has been studied extensively. However, none of the proposed approaches is directly extendable for the efficient support of inverse queries when $|Q| > 1$. First, there exists no related work on reverse queries for the ε-range query predicate. This is not surprising since the the reverse ϵ-range query is equal to a (normal) ε-range query. However, there exists a large body of work for reverse k-nearest neighbor (Rk-NN) queries. Self-pruning approaches like the RNN-tree [6] and the RdNN-tree [14] operate on top of a spatial index, like the R-tree. Their objective is to estimate the k-NN distance of each index entry e. If the k-NN distance of e is smaller than the distance of e to the query q, then e can be pruned. These methods suffer from the high materialization and maintenance cost of the k-NN distances.

Mutual-pruning approaches such as [10–12] use other points to prune a given index entry e. TPL [12] is the most general and efficient approach. It uses 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 using the retrieved neighbors. TPL can be used for inverse k-NN queries where $|Q| > 1$, by simply performing a reverse k-NN query for each query point and then intersecting the results (i.e., the brute-force approach).

For reverse dynamic skyline queries, [3] proposed an efficient solution, which first performs a filter-step, pruning database objects that are globally dominated by some point in the database. For the remaining points, a window query is performed in a refinement step. In addition, [7] gave a solution for reverse dynamic skyline computation on uncertain data. None of these methods considers the case of $|Q| > 1$, which is the focus of our work.

In [13], the problem of reverse top-k queries is studied. A reverse top-k query returns, for a point q and a positive integer k, the set of linear preference functions for which q is contained in their top-k result. The authors provide an efficient solution for the 2D case and discuss its generalization to the multidimensional case, but do not consider the case where $|Q| > 1$. Although we do not study inverse top-k queries in this paper, we note that it is an interesting subject for future work.

Inverse queries are very related to group queries, i.e. similarity queries that retrieve the top-k objects according to a given similarity (distance) aggregate w.r.t. a given set of query points [9, 8]. However, the problem addressed by group queries generally differs from the problem addressed in this paper. Instead of minimizing distance aggregations, here we have to find efficient methods for converging query predicate evaluations w.r.t. a set of query points. Hence, new strategies are required.

3 Inverse Query (IQ) Framework

Our solutions for the three inverse queries under study are based on a common framework consisting of the following filter-refinement pipeline:

Filter 1: Fast Query Based Validation: The first component of the framework, called *fast query based validation*, uses the set of query objects Q only to perform a quick check on whether it is possible to have any result at all. In particular, this filter verifies simple constraints that are necessary conditions for a non-empty result. For example, for the Ik-NN case, the result is empty if $|Q| > k$.

Filter 2: Query Based Pruning: *Query based pruning* again uses the query objects only to prune objects in $\mathcal{D}$ which may not participate in the result. Unlike the simple first filter, here we employ the topology of the query objects.

Filters 1 and 2 can be performed very fast because they do not involve any database object except the query objects.

Filter 3: Object Based Pruning: This filter, called *object based pruning*, is more advanced because it involves database objects additional to the query objects. The strategy is to access database objects in ascending order of their maximum distance to any query point; formally:

$$MaxDist(o, Q) = \max_{q \in Q}(d(e, q)).$$

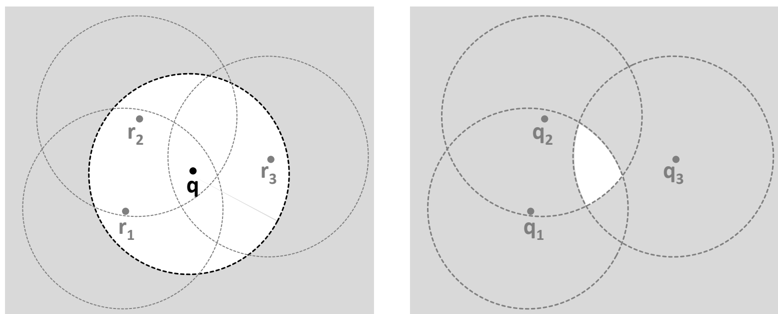
The rationale for this access order is that, given any query object q, objects that are close to q have more pruning power, i.e., they are more likely to prune other objects w.r.t. q than objects that are more distant to q. To maximize the pruning power, we prefer to examine objects that are close to all query points first.

Note that the applicability of the filters depends on the query. *Query based pruning* is applicable if the query objects suffice to restrict the search space which holds for the inverse ε-range query and the inverse skyline query but not directly for the inverse k-NN query. In contrast, the *object based pruning* filter is applicable for queries where database objects can be used to prune other objects which for example holds for the inverse k-NN query and the inverse skyline query but not for the inverse ε-range query.

Refinement: In the final *refinement* step, the remaining candidates are verified and the *true hits* are reported as results.

4 Inverse ε-Range Query

We will start with the simpler query, the inverse ε-range query. First, consider the case of a query object q (i.e., $|Q| = 1$). In this case, the inverse ε-range query computes all objects, that have q within their ε-range sphere. Due to the symmetry of the ε-range query predicate, all objects satisfying the inverse ε-range query predicate are within the ε-range sphere of q as illustrated in Figure 2(a). In the following, we consider the general case, where $|Q| > 1$ and show how our framework can be applied.



(a) Single query case. (b) Multiple query case.

Fig. 2. Pruning space for $I\varepsilon$-RQ.

4.1 Framework Implementation

Fast Query Based Validation: There is no possible result if there exists a pair q, q' of queries in Q, such that their ε-ranges do not intersect (i.e., $d(q, q') > 2 \cdot \varepsilon$). In this case, there can be no object r having both q and q' within its ε-range (a necessary condition for r to be in the result).

Query Based Pruning: Let $S_i^\varepsilon \subseteq \mathbb{R}^d$ be the ε-sphere around query point q_i for all $q_i \in Q$, as depicted in the example shown in Figure 2(b). Obviously, any point in the intersection region of all spheres, i.e. $\cap_{i=1..m} S_i^\varepsilon$, has all query objects $q_i \in Q$ in its ε-range. Consequently, all objects outside of this region can be pruned. However, the computation of the search region can become too expensive in an arbitrary high dimensional space; thus, we compute the intersection between rectangles that minimally bound the hyper-spheres and use it as a filter. This can be done quite efficiently even in high dimensional spaces; the resulting filter rectangle is used as a window query and all objects in it are passed to the refinement step as candidates.

Object Based Pruning: As mentioned in Section 3 this filter is not applicable for inverse ε-range queries, since objects cannot be used to prune other objects.

Refinement: In the refinement step, for all candidates we compute their distances to all query points $q \in Q$ and report only objects that are within distance ε from all query objects.

4.2 Algorithm

The implementation of our framework above can be easily converted to an algorithm, which, after applying the filter steps, performs a window query to retrieve the candidates, which are finally verified. Search can be facilitated by an R-tree that indexes $\mathcal{D}$. Starting from the root, we search the tree, using the filter rectangle. To minimize the I/O cost, for each entry P of the tree that intersects the filter rectangle, we compute its distance to all points in Q and access the corresponding subtree only if all these distances are smaller than ε.

5 Inverse *k*-NN Query

For inverse k-nearest neighbor queries (Ik-NNQ), we first consider the case of a single query object (i.e., $|Q| = 1$). As discussed in Section 2, this case can be processed by the bi-section-based Rk-NN approach (TPL) proposed in [12], enhanced by the rectangle-based pruning criterion proposed in [4]. The core idea of TPL is to use bi-section-hyperplanes between database objects o and the query object q in order to check which objects are closer to o than to q. Each bi-section-hyperplane divides the object space into two half-spaces, one containing q and one containing o. Any object located in the half-space containing o is closer to o than to q. The objects spanning the hyperplanes are collected in an iterative way. Each object o is then checked against the resulting half-spaces that do not contain q. As soon as o is inside more than k such half-spaces, it can be pruned. Next, we consider queries with multiple objects (i.e., $|Q| > 1$) and discuss how the framework presented in Section 3 is implemented in this case.

5.1 Framework Implementation

Fast Query Based Validation Recall that this filter uses the set of query objects Q only, to perform a quick check on whether the result is empty. Here, we use the obvious rule that the result is empty if the number of query objects exceeds query parameter k.

Query Based Pruning We can exploit the query objects in order to reduce the Ik-NN query to an Ik'-NN query with $k' < k$. A smaller query parameter k' allows us to terminate the query process earlier and reduce the search space. We first show how k can be reduced by means of the query objects only. The proofs for all lemmas can be found in the full version of this paper [2].

Lemma 1. *Let $\mathcal{D} \subseteq \mathbb{R}^d$ be a set of database objects and $Q \subseteq \mathcal{D}$ be a set of query objects. Let $\mathcal{D}' = \mathcal{D} - Q$. For each $o \in \mathcal{D}'$, the following statement holds:*

$$o \in Ik\text{-}NNQ(Q) \text{ in } \mathcal{D} \Rightarrow \forall q \in Q : o \in Ik'\text{-}NNQ(\{q\}) \text{ in } \mathcal{D}' \cup \{q\},$$

$$\text{where } k' = k - |Q| + 1.$$

Simply speaking, if a candidate object o is not in the Ik'-$NNQ(\{q\})$ result of some $q \in Q$ considering only the points $\mathcal{D}' \cup \{q\}$, then o cannot be in the Ik-$NNQ(Q)$ result considering all points in $\mathcal{D}$ and o can be pruned. As a consequence, Ik'-$NNQ(\{q\})$ in $\mathcal{D}' \cup \{q\}$ can be used to prune candidates for any $q \in Q$. The pruning power of Ik'-$NNQ(\{q\})$ depends on how $q \in Q$ is selected.

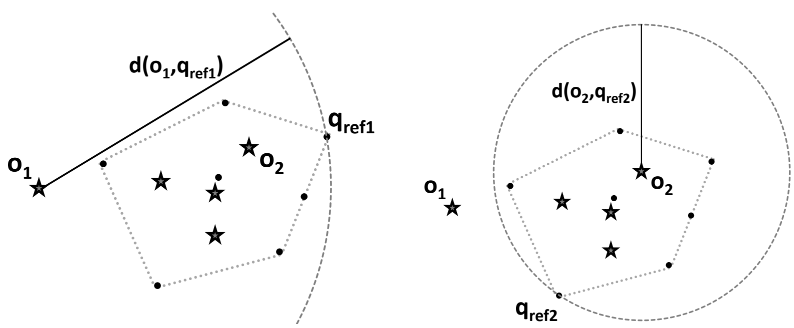
From Lemma 1 we can conclude the following:

Lemma 2. *Let $o \in \mathcal{D} - Q$ be a database object and $q^o_{ref} \in Q$ be a query object such that $\forall q \in Q : d(o, q^o_{ref}) \geq d(o, q)$. Then*

$$o \in Ik\text{-}NNQ(Q) \Leftrightarrow o \in Ik'\text{-}NNQ(\{q^o_{ref}\}) \text{ in } \mathcal{D}' \cup \{q\},$$

where $k' = k - |Q| + 1$.

Lemma 2 suggests that for any candidate object o in $\mathcal{D}$, we should use the farthest query point to check whether o can be pruned.



(a) Pruning o_1 (b) Pruning o_2

Fig. 3. Ik-NN pruning based on Lemma 4

Object Based Pruning Up to now, we only used the query points in order to reduce k in the inverse k-NN query. Now, we will show how to consider database objects in order to further decrease k.

Lemma 3. *Let Q be the set of query objects and $\mathcal{H} \subseteq \mathcal{D} - Q$ be the non-query(database) objects covered by the convex hull of Q. Furthermore, let $o \in \mathcal{D}$ be a database object and $q^o_{ref} \in Q$ a query object such that $\forall q \in Q : d(o, q^o_{ref}) \geq d(o, q)$. Then for each object $p \in \mathcal{H}$ it holds that $d(o, p) \leq d(o, q^o_{ref})$.*

According to the above lemma the following statement holds:

Lemma 4. *Let Q be the set of query objects, $\mathcal{H} \subseteq \mathcal{D} - Q$ be the database (non-query) objects covered by the convex hull of Q and let $q^o_{ref} \in Q$ be a query object such that $\forall q \in Q : d(o, q^o_{ref}) \geq d(o, q)$. Then for a given database object $o \in \mathcal{D}$*

$$\forall o \in \mathcal{D} - \mathcal{H} - Q : o \in Ik\text{-}NNQ(Q) \Leftrightarrow$$

at most $k' = k - |\mathcal{H}| - |Q|$ *objects* $p \in \mathcal{D} - \mathcal{H}$ *are closer to* o *than* q^o_{ref}, *and*

$$\forall o \in \mathcal{H} : o \in Ik\text{-}NNQ(Q) \Leftrightarrow$$

at most $k' = k - |\mathcal{H}| - |Q| + 1$ *objects* $p \in \mathcal{D} - \mathcal{H}$ *are closer to* o *than* q^o_{ref}.

Based on Lemma 4, given the number of objects in the convex hull of Q, we can prune objects outside of the hull from Ik-NN(Q). Specifically, for an Ik-NN query we have the following pruning criterion: An object $o \in \mathcal{D}$ can be pruned, as soon as we find more than k' objects $p \in \mathcal{D} - \mathcal{H}$ outside of the convex hull of Q, that are closer to o than q^o_{ref}. Note that the parameter k' is set according to Lemma 4 and depends on whether o is in the convex hull of Q or not. Depending on the size of Q and the number of objects within the convex hull of Q, $k' = k - |\mathcal{H}| + 1$ can become negative. In this case, we can terminate query evaluation immediately, as no object can qualify the inverse query (i.e., the inverse query result is guaranteed to be empty). The case where $k' = k - |\mathcal{H}| + 1$ becomes zero is another special case, as all objects outside of $\mathcal{H}$ can be pruned. For all objects in the convex hull of Q (including all query objects) we have to check whether there are objects outside of $\mathcal{H}$ that prune them.

As an example of how Lemma 4 can be used, consider the data shown in Fig. 3 and assume that we wish to perform an inverse 10NN query using a set Q of seven query objects, shown as points in the figure; non-query database points are represented by stars. In Figure 3(a), the goal is to determine whether candidate object o_1 is a result, i.e., whether o_1 has all $q \in Q$ in its 10NN set. The query object having the largest distance to o_1 is q_{ref1}. Since o_1 is located outside of the convex hull of Q (i.e, $o \in \mathcal{D} - \mathcal{H} - Q$), the first equivalence of Lemma 4, states that o_1 is a result if at most $k' = k - |\mathcal{H}| - |Q| = 10 - 4 - 7 = -1$ objects in $\mathcal{D} - \mathcal{H} - Q$ are closer to o_1 than q_{ref1}. Thus, o_1 can be safely pruned without even considering these objects (since obviously, at least zero objects are closer to o_1 than q_{ref1}). Next, we consider object o_2 in Figure 3(b). The query object with the largest distance to o_2 is q_{ref2}. Since o_2 is inside the convex hull of Q, the second equivalence of Lemma 4 yields that o_2 is a result if at most $k' = k - |\mathcal{H}| - |Q| + 1 = 10 - 4 - 7 + 1 = 0$ objects $\mathcal{D} - \mathcal{H} - Q$ are closer to o_2 than q_{ref2}. Thus, o_2 remains a candidate until at least one object in $\mathcal{D} - \mathcal{H} - Q$ is found that is closer to o_2 than q_{ref2}.

Refinement Each remaining candidate is checked whether it is a result of the inverse query by performing a k-NN search and verifying whether its result includes Q.

5.2 Algorithm

We now present a complete algorithm that traverses an *aggregate* R-tree ($ARTree$), which indexes $\mathcal{D}$ and computes $Ik\text{-}NNQ(Q)$ for a given set Q of query objects, using Lemma 4 to prune the search space. The entries in the tree nodes are augmented with the cardinality of objects in the corresponding sub-tree. These counts can be used to accelerate search, as we will see later.

Algorithm 1 Inverse kNNQuery

```
Require: Q, k, ARTree
//Fast Query Based Validation
if |Q| > k then
    return "no result" and terminate algorithm
end if
pq PriorityQueue ordered by max_{q_i ∈ Q} MinDist
pq.add(ARTree.root entries)
|H| = 0
LIST candidates, prunedEntries
//Query/Object Based Pruning
while ¬pq.isEmpty() do
    e = pq.poll()
    if getPruneCount(e, Q, candidates, prunedEntries, pq) > k − |H| − |Q| then
        prunedEntries.add(e)
    else if e.isLeafEntry() then
        candidates.add(e)
    else
        pq.add(e.getChildren())
    end if
    if e ∈ convexHull(Q) then
        |H| += e.agg_count
    end if
end while
//Refinement Step
LIST result
for c ∈ candidates do
    if q^o_ref ∈ knnQuery(c, k) then
        result.add(c)
    end if
end for
return (result)
```

In a nutshell, the algorithm, while traversing the tree, attempts to prune nodes based on the lemma using the information known so far about the points of $\mathcal{D}$ that are included in the convex hull (*filtering*). The objects that survive the pruning are inserted in the *candidates* set. During the *refinement* step, for each point c in the candidates set, we run a k-NN query to verify whether c contains Q in its k-NN set.

Algorithm 1 is a pseudocode of our approach. The $ARTree$ is traversed in a best-first search manner [5], prioritizing the access of the nodes according to the maximum possible distance (in case of a non-leaf entry we use *MinDist*) of their contents to the query points Q. In specific, for each R-tree entry e we can compute, based on its MBR, the farthest possible point q^o_{ref} in Q to a point p indexed under e. Processing the entries with the smallest such distances first helps to find points in the convex hull of Q earlier, which helps making the pruning bound tighter.

Thus, initially, we set $|\mathcal{H}| = 0$, assuming that in the worst case the number of non-query points in the convex hull of Q is 0. If the object which is deheaped is inside the convex hull, we increase $|\mathcal{H}|$ by one. If a non-leaf entry is deheaped and its MBR is contained in the hull, we increase $|\mathcal{H}|$ by the number of objects in the corresponding sub-tree, as indicated by its augmented counter.

During tree traversal, the accessed tree entries could be in one of the following sets (i) the set of *candidates*, which contains objects that could possibly be results of the inverse query, (ii) the set of *pruned entries*, which contains (pruned) entries whose sub-trees may not possibly contain inverse query results, and (iii) the set of entries which are currently in the priority queue. When an entry e is deheaped, the algorithm checks whether it can be pruned. For this purpose, it initializes a $prune\_counter$ which is a lower bound of the number of objects that are closer to every point p in e than Q's farthest point to p. For every entry e' in all three sets (candidates, pruned, and priority queue), we increase the $prune\_counter$ of e by the number of points in e' if the following condition holds: $\forall p \in e, \forall p' \in e' : dist(e, e') < dist(e, q^o_{ref})$. This condition can efficiently be checked [4]. An example where this condition is fulfilled is shown in Figure 4. Here the $prune\_counter$ of e can be increased by the number of points in e'.

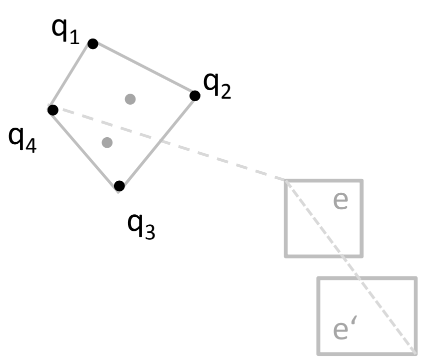


Fig. 4. Calculating the $prune\_count$ of e

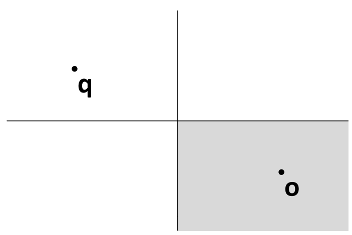
While updating $prune\_counter$ for e, we check whether $prune\_counter > k - |\mathcal{H}| - |Q|$ ($prune\_counter > k - |\mathcal{H}| - |Q| + 1$) for entries that are entirely outside of (intersect) the convex hull. As soon as this condition is true, e can be pruned as it cannot contain objects that can participate in the inverse query result (according to Lemma 4). Considering again Figure 4 and assuming the number of points in e' to be 5, e could be pruned for $k \leq 10$ (since $prune\_counter(5) > k(10) - |\mathcal{H}|(2) - |Q|(4)$ holds). In this case e is moved to the set of pruned entries. If e survives pruning, the node pointed to by e is visited and its entries are enheaped if e is a non-leaf entry; otherwise e is inserted in the *candidates* set. When the queue becomes empty, the filter step of the algorithm completes with a set of *candidates*. For each object c in this set, we check whether c is a result of the inverse query by performing a k-NN search and verifying whether its result includes Q. In our implementation, to make this test faster, we replace the k-NN search by an aggregate ε-range query around c, by setting $\varepsilon = d(c, q^c_{ref})$. The objective is to count whether the number of objects in the range is greater than k. In this case, we can prune c, otherwise c is a result of the inverse query. $ARTree$ is used to process the aggregate ε-range query; for every entry e included in the ε-range, we just increase the aggregate count by the augmented counter to e without having to traverse the corresponding subtree. In addition, we perform batch searching for candidates that are close to each other, in order to optimize performance. The details are skipped due to space constraints.

6 Inverse Dynamic Skyline Query

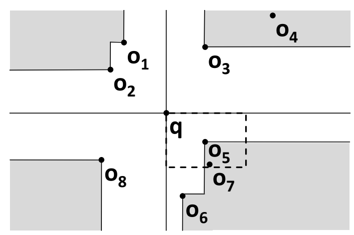
We again first discuss the case of a single query object, which corresponds to the reverse dynamic skyline query [7] and then present a solution for the more interesting case where $|Q| > 1$. Let q be the (single) query object with respect to which we want to compute the inverse dynamic skyline. Any object $o \in \mathcal{D}$ defines a pruning region, such that any object o' in this region cannot be part of the inverse query result. Formally:

Definition 2 (Pruning Region). *Let $q = (q^1, \ldots, q^d) \in Q$ be a single d-dimensional query object and $o = (o^1, \ldots, o^d) \in \mathcal{D}$ be any d-dimensional database object. Then the pruning region $PR_q(o)$ of o w.r.t. q is defined as the d-dimensional rectangle where the ith dimension of $PR_q(o)$ is given by $[\frac{q^i+o^i}{2}, +\infty]$ if $q^i \leq o^i$ and $[-\infty, \frac{q^i+o^i}{2}]$ if $q^i \geq o^i$.*

The pruning region of an object o with respect to a single query object q is illustrated by the shaded region in Figure 5(a).



(a) pruning region



(b) candidates

Fig. 5. Single-query case

Filter step. As shown in [7], any object $p \in \mathcal{D}$ can be safely pruned if p is contained in the pruning region of some $o \in \mathcal{D}$ w.r.t. q (i.e. $p \in PR_q(o)$). Accordingly, we can use q to divide the space into 2^d partitions by splitting along each dimension at q. Let $o \in \mathcal{D}$ be an object in any partition P; o is an *I-DSQ candidate*, iff there is no other object $p \in P \subseteq \mathcal{D}$ that dominates o w.r.t. q.

Thus, we can derive all I-DSQ candidates as follows: First, we split the data space into the 2^d partitions at the query object q as mentioned above. Then in each partition, we compute the skyline[3], as illustrated in the example depicted in Figure 5(b). The union of the four skylines is the set of the inverse query candidates (e.g., $\{o_1, o_2, o_3, o_5, o_6, o_8\}$ in our example).

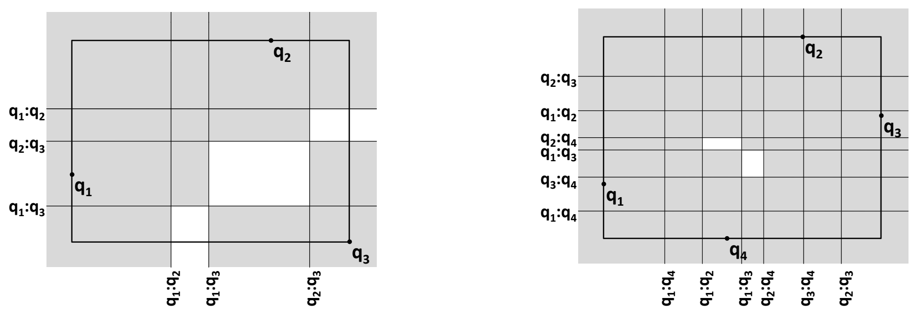
Refinement. The result of the reverse dynamic skyline query is finally obtained by verifying for each candidate c, whether there is an object in $\mathcal{D}$ which dominates q w.r.t. c. This can be done by checking whether the hypercube centered at c with extent $2 \cdot |c^i - q^i|$ at each dimension i is empty. For example, candidate o_5 in Figure 5(b) is not a result, because the corresponding box (denoted by dashed lines) contains o_7. This means that in both dimensions o_7 is closer to o_5 than q is.

[3] Only objects within the same partition are considered for the dominance relation.

6.1 IQ Framework Implementation

Fast Query Based Validation Following our framework, first the set Q of query objects is used to decide whether it is possible to have any result at all. For this, we use the following lemma:

Lemma 5. *Let $q \in Q$ be any query object and let $\mathcal{S}$ be the set of 2^d partitions derived from dividing the object space at q along the axes into two halves in each dimension. If in each partition $r \in \mathcal{S}$ there is at least one query object $q' \in Q$ ($q' \neq q$), then there cannot be any result.*



(a) $|Q| = 3$ (b) $|Q| = 4$

Fig. 6. Pruning regions of query objects

Query Based Pruning We now propose a filter, which uses the set Q of query objects only in order to reduce the space of candidate results. We explore similar strategies as the fast query based validation. For any pair of query objects $q, q' \in Q$, we can define two pruning regions, according to Definition 2: $PR_q(q')$ and $PR_{q'}(q)$. Any object inside these regions cannot be a candidate of the inverse query result because it cannot have both q_1 and q_2 in its dynamic skyline point set. Thus, for every pair of query objects, we can determine the corresponding pruning regions and use their union to prune objects or R-tree nodes that are contained in it. Figure 6 shows examples of the pruning space for $|Q| = 3$ and $|Q| = 4$. Observe that with the increase of $|Q|$ the remaining space, which may contain candidates, becomes very limited.

The main challenge is how to encode and use the pruning space defined by Q, as it can be arbitrarily complex in the multidimensional space. As for the Ik-NNQ case, our approach is not to explicitly compute and store the pruning space, but to check on-demand whether each object (or R-tree MBR) can be pruned by one or more query pairs. This has a complexity of $O(|Q|^2)$ checks per object. In the full version of the paper [2], we show how to reduce this complexity for the special 2D case. The techniques shown there can also be used in higher dimensional spaces, with lower pruning effect.

Object Based Pruning For any candidate object o that is not pruned during the query-based filter step, we need to check if there exists any other database object o' which dominates some $q \in Q$ with respect to o. If we can find such an o', then o cannot have q in its dynamic skyline and thus o can be pruned for the candidate list.

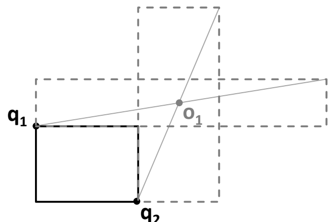


Fig. 7. Refinement area defined by q_1, q_2 and o_1

Refinement In the refinement step, each candidate c is verified by performing a dynamic skyline query using c as query point. The result should contain all $q_i \in Q$, otherwise c is dropped. The refinement step can be improved by the following observation (cf. Figure 7): for checking if a candidate o_1 has all $q_i \in Q$ in its dynamic skyline, it suffices to check whether there exists at least one other object $o_j \in \mathcal{D}$ which prevents one q_i from being part of the skyline. Such an object has to lie within the MBR defined by q_i and q_i' (which is obtained by reflecting q_i through o_1). If no point is within the $|Q|$ MBRs, then o_1 is reported as result.

6.2 Algorithm

The algorithm for $I\text{-}DSQ$, during the filter step, traverses the tree in a best first manner, where entries are accessed by their minimal distance (MinDist) to the farthest query object. For each entry e we check if e is completely contained in the union of pruning regions defined by all pairs of queries $(q_i, q_j) \in Q$; i.e., $\bigcup_{(q_i,q_j)\in Q} PR_{q_i}(q_j)$. In addition, for each accessed database object o_i and each query object q_j, the pruning region is extended by $PR_{q_j}(o_i)$. Analogously to the $Ik\text{-}NN$ case, lists for the candidates and pruned entries are maintained. Finally, the remaining candidates are refined using the refinement strategy described in Section 6.1.

7 Experiments

For each of the inverse query predicates discussed in the paper, we compare our proposed solution based on multi-query-filtering (MQF), with a naive approach (Naive) and another intuitive approach based on single-query-filtering (SQF). The naive algorithm (Naive) computes the corresponding reverse query for every $q \in Q$ and intersects their results iteratively. To be fair, we terminated Naive as soon as the intersection of results obtained so far is empty. SQF performs a Rk-NN (Rε-range / RDS) query using one randomly chosen query point as a filter step to obtain candidates. For each candidate an ε-range (k-NN / DS) query is issued and the candidate is confirmed if all query points are contained in the result of the query (refinement step). Since the pages accessed by the queries in the refinement step are often redundant, we use a buffer to further boost the performance of SQF. We employed R^*-trees ([1]) of pagesize 1Kb to index the data sets used in the experiments. For each method, we present the number of page accesses and runtime. To give insights into the impact of the different parameters on the cardinality of the obtained results we also included this number to the charts. In all settings we performed 1000 queries and averaged the results. All methods were implemented in Java 1.6 and tests were run on a dual core (3.0 Ghz) workstation with 2 GB main memory having windows xp as OS. The performance evaluation settings are summarized below; the numbers in **bold** correspond to the default settings:

parameter	values
db size	100000 (synthetic), 175812 (real)
dimensionality	2, **3**, 4, 5
ε	0.04, 0.05, **0.06**, 0.07, 0.08, 0.09, 0.1
k	50, **100**, 150, 200, 250
# inverse queries	1, 3, 5, **10**, 15, 20, 25, 30, 35
query extent	0.0001, 0.0002, 0.0003, **0.0004**, 0.0005, 0.0006

The experiments were performed using several data sets:

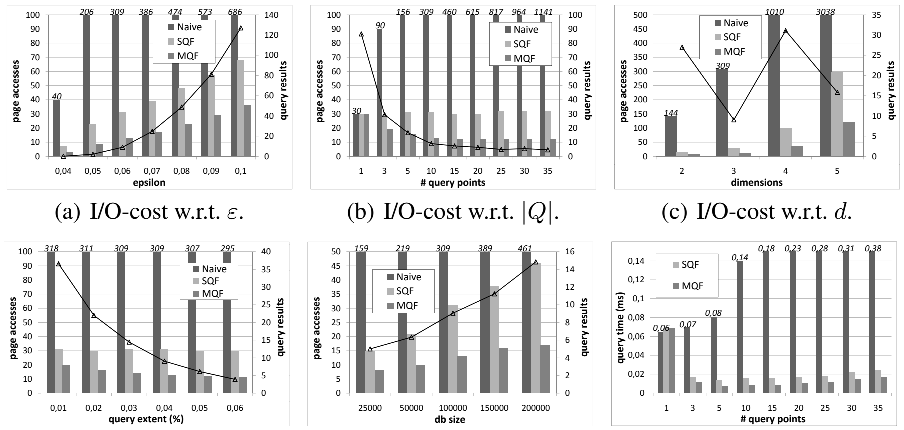
- Synthetic data sets: Clustered and uniformly distributed objects in d-dimensional space.
- Real Data set: Vertices in the Road Network of North America [4]. Contains 175,812 two-dimensional points.

The data sets were normalized, such that their minimum bounding box is $[0, 1]^d$. For each experiment, the query objects Q for the inverse query were chosen randomly from the database. Since the number of results highly depends on the distance between inverse query points (in particular for the $I\varepsilon$-RQ and Ik-NNQ) we introduced an additional parameter called *extent* to control the maximal distance between the query objects. The value of *extent* corresponds to the volume (fraction of data space) of a cube that minimally bounds all queries. For example in the 3D space the default cube would have a side length of 0.073. A small *extent* assures that the queries are placed close to each other generally resulting in more results. In this section, we show the behavior of all three algorithms on the uniform data sets only. Experiments on the other data sets can be found in the full version of the paper [2].

7.1 Inverse ε-Range Queries

We first compared the algorithms on inverse ε range queries. Figure 8(a) shows that the relative speed of our approach (MQF) compared to Naive grows significantly with increasing ε; for Naive, the cardinality of the result set returned by each query depends on the space covered by the hypersphere which is in $O(\varepsilon^d)$. In contrast, our strategy applies spatial pruning early, leading to a low number of page accesses. SQF is faster than Naive, but still needs around twice as much page accesses as MQF. MQF performs even better with an increasing number of query points in Q (as depicted in Figure 8(b)), as in this case the intersection of the ranges becomes smaller. The I/O-cost of SQF in this case remains almost constant which is mainly due to the use of the buffer which lowers the page accesses in the refinement step. Similar results can be observed when varying the database size (Figure 8(e)) and query extent (Figure 8(d)). For the data dimensionality experiment (Figure 8(c)) we set epsilon such that the sphere defined by ε covers always the same percentage of the dataspace, to make sure that we still obtain results when increasing the dimensionality (note, however, that the number of results is still unsteady). Increasing dimensionality has a negative effect on performance. However

[4] Obtained and modified from *http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm*. The original source is the *Digital Chart of the World Server (http://www.maproom.psu.edu/dcw/)*.



(a) I/O-cost w.r.t. ε. (b) I/O-cost w.r.t. $|Q|$. (c) I/O-cost w.r.t. d.

(d) I/O-cost w.r.t. extent. (e) I/O-cost w.r.t. $|\mathcal{D}|$. (f) CPU-cost w.r.t. $|Q|$.

Fig. 8. $I\varepsilon$-Q algorithms on uniform data set

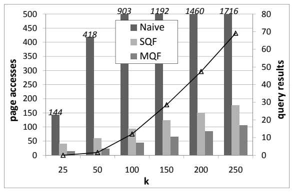
MQF copes better with data dimensionality than the other approaches. Finally, Figure 8(f) compares the computational costs of the algorithms. Even though Inverse Queries are I/O bound, MQF is still preferable for main-memory problems.

7.2 Inverse k-NN Queries

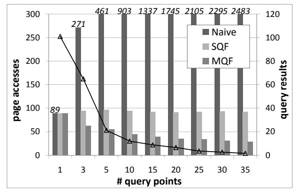
The three approaches for inverse k-NN search show a similar behavior as those for the Iε-RQ. Specifically the behavior for varying k (Figure 9(a)) is comparable to varying ε and increasing the query number (Figure 9(b)) and the query extent (Figure 9(d)) yields the expected results. When testing on data sets with different dimensionality, the advantage of MQF becomes even more significant when d increases (cf. Figure 9(c)). In contrast to the Iε-RQ results for Ik-NN queries the page accesses of MQF decrease (see Figure 9(e)) when the database size increases (while the performance of SQF still degrades). This can be explained by the fact, that the number of pages accessed is strongly correlated with the number of obtained results. Since for the Iε-RQ the parameter ε remained constant, the number of results increased with a larger database. For Ik-NN the number of results in contrast decreases and so does the number of accessed pages by MQF. As in the previous set of experiments MQF has also the lowest runtime (Figure 9(f)).

7.3 Inverse Dynamic Skyline Queries

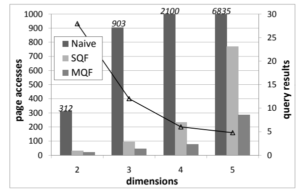
Similar results as for the Ik-NNQ algorithm are obtained for the inverse dynamic skyline queries (I-DSQ). Increasing the number of queries in Q reduces the cost of the MQF approach, while the costs of the competitors increase. Since the average number of results approaches 0 faster than for the other two types of inverse queries we choose 4 as the default size of the query set. Note that the number of results for I-DSQ intuitively increases exponentially with the dimensionality of the data set (cf. Figure 10(b)),



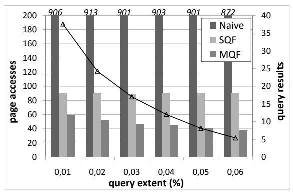
(a) I/O-cost w.r.t. k.



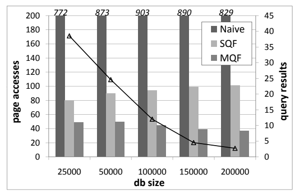
(b) I/O-cost w.r.t. $|Q|$.



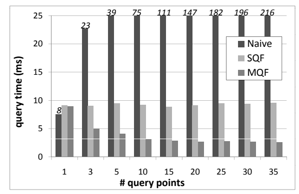
(c) I/O-cost w.r.t. d.



(d) I/O-cost w.r.t. extent.



(e) I/O-cost w.r.t. $|\mathcal{D}|$.



(f) CPU-cost w.r.t. $|Q|$.

Fig. 9. Ik-NNQ algorithms on uniform data set

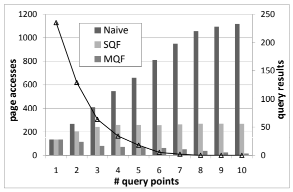
thus this value can be much larger for higher dimensional data sets. Increasing the distance among queries does not affect the performance as seen in Figure 10(c); regarding the number of results in contrast to inverse range- and k-NN queries, inverse dynamic skyline queries are almost insensitive to the distance among the query points. The rationale is that dynamic skyline queries can have results which are arbitrary far from the query point, thus the same holds for the inverse case. The same effect can be seen for increasing database size (cf. Figure 10(d)). The advantage of MQF remains constant over the other two approaches. Like inverse range and k-NN queries, I-DSQ are I/O bound (see Figure 10(e)), but MQF is still preferable for main-memory problems.

8 Conclusions

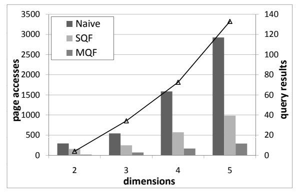
In this paper we introduced and formalized the problem for inverse query processing. We proposed a general framework to such queries using a filter-refinement strategy and applied this framework to the problem of answering inverse ε-range queries, inverse k-NN queries and inverse dynamic skyline queries. Our experiments show that our framework significantly reduces the cost of inverse queries compared to straightforward approaches. In the future, we plan to extend our framework for inverse queries with different query predicates, such as top-k queries. In addition, we will investigate inverse query processing in the bi-chromatic case, where queries and objects are taken from different data sets. Another interesting extension of inverse queries is to allow the user not only to specify objects that have to be in the result, but also objects that must not be in the result.

Acknowledgements

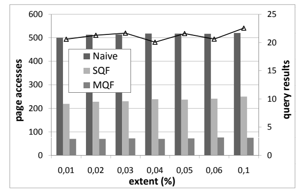
This work was supported by a grant from the Germany/Hong Kong Joint Research Scheme sponsored by the Research Grants Council of Hong Kong (Reference No. G_HK030/09) and the Germany Academic Exchange Service of Germany (Proj. ID 50149322).



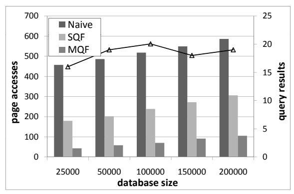
(a) I/O-cost w.r.t. $|Q|$.



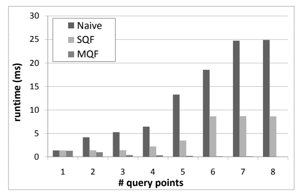
(b) I/O-cost w.r.t. d.



(c) I/O-cost w.r.t. extent.



(d) I/O-cost w.r.t. $|\mathcal{D}|$.



(e) CPU-cost w.r.t. $|Q|$.

Fig. 10. $I\text{-}DSQ$ algorithms on uniform data set

References

1. 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.
2. T. Bernecker, T. Emrich, H.-P. Kriegel, N. Mamoulis, S. Zhang, M. Renz, and A. Züfle. Inverse queries for multidimensional spaces. In *The ACM Computing Research Repository (CoRR), http://arxiv.org/abs/1103.0172*, 2011.
3. E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, pages 291–302, 2007.
4. T. Emrich, H.-P. Kriegel, P. Kröger, M. Renz, and A. Züfle. Boosting spatial pruning: On optimal pruning of mbrs. In *SIGMOD, June 6-11, 2010*.
5. G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Proc. SSD*, 1995.
6. F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. SIGMOD*, 2000.
7. X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD Conference*, pages 213–226, 2008.
8. D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 301–312, 2004.
9. D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576, 2005.
10. A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High dimensional reverse nearest neighbor queries. In *Proc. CIKM*, 2003.
11. I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *Proc. DMKD*, 2000.
12. Y. Tao, D. Papadias, and X. Lian. Reverse kNN search in arbitrary dimensionality. In *Proc. VLDB*, 2004.
13. A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE*, pages 365–376, 2010.
14. C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. ICDE*, 2001.