Geoinformatica (2013) 17:449–487
DOI 10.1007/s10707-012-0162-y

Spatial inverse query processing

Thomas Bernecker · Tobias Emrich · Hans-Peter Kriegel · Nikos Mamoulis · Matthias Renz · Shiming Zhang · Andreas Züfle

Received: 30 January 2012 / Revised: 4 June 2012 /
Accepted: 12 July 2012 / Published online: 24 August 2012

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

T. Bernecker · T. Emrich · H.-P. Kriegel · M. Renz · A. Züfle (✉)
Institute for Informatics, Ludwig-Maximilians-Universität München,
Oettingenstr. 67, 80538 München, Germany
e-mail: zuefle@dbs.ifi.lmu.de

T. Bernecker
e-mail: bernecker@dbs.ifi.lmu.de

T. Emrich
e-mail: emrich@dbs.ifi.lmu.de

H.-P. Kriegel
e-mail: kriegel@dbs.ifi.lmu.de

M. Renz
e-mail: renz@dbs.ifi.lmu.de

N. Mamoulis · S. Zhang
Department of Computer Science, University of Hong Kong,
Pokfulam Road, Hong Kong, Hong Kong

N. Mamoulis
e-mail: nikos@cs.hku.hk

S. Zhang
e-mail: smzhang@cs.hku.hk

queries. Furthermore, we show how to relax the definition of inverse queries in order to ensure non-empty result sets. Our experiments show that our framework is significantly more efficient than naive approaches.

Keywords Spatial data · Inverse queries · k-NN query · Skyline query · Range query · Reverse queries

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, w.r.t. a given query predicate. A characteristic example is the reverse k-NN query [6, 14], whose objective is to find the query objects (from a given dataset) 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 query objects, having a given set of data objects within their k-NN result. Besides k-NN search, reverse queries have also been studied for other spatial and multidimensional search problems, such as top-k search [16] 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. Korn and Muthukrishnan [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 each given by a set of metric features values (e.g. price, weight, rating, etc.) 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, i.e. 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. As a spatial application consider a pizza restaurant that wants to find all customers that

are spatially close to both him and another competitor pizza restaurant, in order to find households to direct advertisements to. Again, the spatially close can refer to direct distance (ε-range) or to nearest-neighbor distance.

More specifically, consider a set $\mathcal{D} \in \mathbb{R}^d$ as a database of n objects. We assume that Euclidean distance is used to measure dissimilarity between database objects.[1] 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:

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

Note, that in the context of inverse queries the term "query object" is used in an opposite sense. Here, $q \in Q$ denotes a query point in the context of "inverse queries" while r denotes a query point in the traditional sense of queries, i.e. q is a result of a query based on a query predicate $\mathcal{P}$ using r as query object. 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

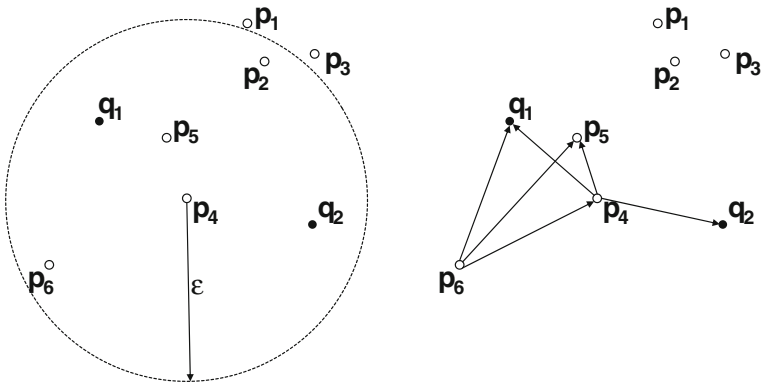
- The mono-chromatic inverse $\mathcal{P}$ query, for which the result set is a subset of $\mathcal{D}$.
- The bi-chromatic inverse $\mathcal{P}$ query, for which the result set is a subset of a given database $\mathcal{D}' \subseteq \mathbb{R}^d$. The main difference with the mono-chromatic inverse $\mathcal{P}$ query is that objects in $\mathcal{D}$ do not affect each other, i.e., when computing the result of an inverse query for a query subset $Q \subseteq \mathcal{D}'$, then only objects in $\mathcal{D}$ are considered for the result computation.

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 query 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 Fig. 1a and the set of database points $\mathcal{D} = \{p_1, p_2, \cdots, p_6, q_1, q_2\}$. 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 $d(p_1, q_2) > \varepsilon$).

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

[1]We note that the proposed techniques can be easily adapted to use any L_p-Norm as well as weighted euclidean distance.



(a) $I\varepsilon$-RQ. (b) Ik-NNQ, $k = 3$.

Fig. 1 Examples of inverse queries

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 100-NN query, using the set of robbed households as Q, returns the set of possible suspects. A mono-chromatic inverse 3-NN query for $Q = \{q_1, q_2\}$ in Fig. 1b returns $\{p_4\}$. p_6, for example, is dropped, as q_2 is not contained in the list of its three 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 to their customers products 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 customers 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 Fig. 1, assuming that $Q = \{q_1, q_2\}$ are clicked products, $I\text{-}DSQ(Q)$ 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.

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. Details of our framework are shown in Section 3. 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 $I\varepsilon$-RQ queries the application of our framework is straightforward, for Ik-NNQ and I-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 Section 2, we give an overview of the previous work which is 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, for both the mono-chromatic and the bi-chromatic case. Section 7 gives a relaxed definition of the problem for the case where no results are obtained using the original definition. In Section 8, we provide an extensive experimental evaluation. Finally, Section 9 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 [17] 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 [12–14] use other points to prune a given index entry e. TPL [14] 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).

Efficient algorithms to process Skyline queries [11, 15] and dynamic skyline queries [2] have gained much attention by scientists due to their manifold applica-

tions in decision making. For reverse dynamic skyline queries, Dellis and Seeger [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, Lian and Chen [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 [16] 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, 10]. 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:

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

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. Furthermore, we note that object based pruning is not applicable for the bi-chromatic case, where objects in $\mathcal{D}$ do not affect each other. For all solutions that we will propose in this work for the mono-chromatic case, the bi-chromatic case can be easily derived by simply disabling object based pruning.

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 Fig. 2a. In the following, we consider the general case, where $|Q| > 1$ and show how our framework can be applied.

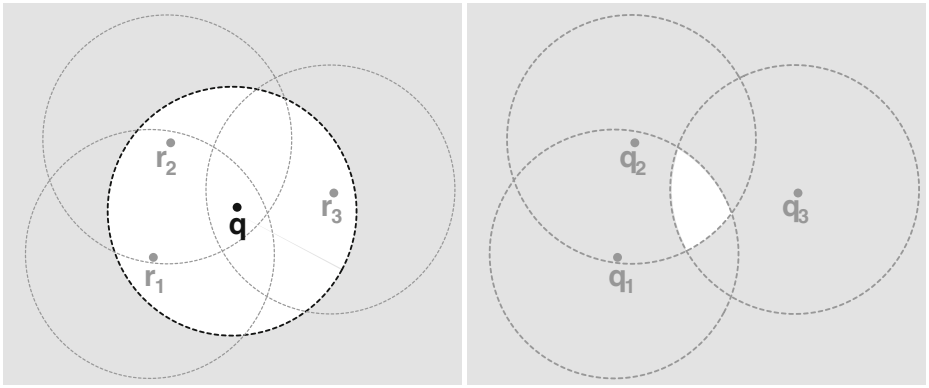
4.1 Framework implementation

4.1.1 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).

4.1.2 Query based pruning

For each $q_i \in Q$, let $S_i^\varepsilon \subseteq \mathbb{R}^d$ denote the ε-sphere around point q_i as depicted in the example shown in Fig. 2b. 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, even for the two-dimensional case where, in the worst case, the search region may consist of $|\mathcal{D}|$ circular boundary lines. Thus, we propose to 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.



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

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

4.1.3 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.

4.1.4 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

Due to the straightforward nature of our *Iε-range* algorithm, we will give only a discuss on issues related with this algorithm in this section, rather than a formal pseudo-code. 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.

For inverse ε-range queries, the bi-chromatic case is identical to the mono-chromatic case. The reason being that non-query database objects in $\mathcal{D}$ do not affect each other. Thus, the predicate of a query object $q \in Q$ being in range of $o \in \mathcal{D}$ only depends on the position of o and q, but not on other objects in $\mathcal{D} \setminus \{o\}$.

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 [14], 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

5.1.1 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 the query parameter k.

5.1.2 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.

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:*

$$o \in Ik\text{-}NNQ(Q) \text{ in } \mathcal{D} \Rightarrow \forall q \in Q : o \in Ik'\text{-}NNQ(\{q\}) \text{ in } \mathcal{D}' \cup \{q\},$$
$$\text{where } k' = k - |Q| + 1.$$

Proof Assume that

$$\forall q \in Q : o \in Ik'\text{-}NNQ(\{q\}) \text{ in } \mathcal{D}' \cup \{q\}$$
$$\text{where } k' = k - |Q| + 1.$$

Then, for each $q \in Q$ there exist at most $k' - 1$ objects $o' \in \mathcal{D} \setminus Q$ such that $dist(o, o') < dist(o, q)$. In addition, there exist at most $|Q| - 1$ query objects $q' \in Q \setminus \{q\}$ such that $dist(o, q') < dist(o, q)$. Thus, there exist at most $k' - 1 + |Q| - 1 = k - |Q| + 1 - 1 + |Q| - 1 = k - 1$ objects which are closer to o than to q, thus q must be a k-NN of o. Since this holds for each $q \in Q$ we obtain

$$o \in Ik\text{-}NNQ(Q) \text{ in } \mathcal{D}$$

□

Lemma 1 implies that, 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 $Ik\text{-}NNQ(Q)$ result considering all points in $\mathcal{D}$ and o can be pruned. As a consequence, $Ik'\text{-}NNQ(\{q\})$ in $\mathcal{D}' \cup \{q\}$ can be used to prune candidates for any $q \in Q$. The pruning power of $Ik'\text{-}NNQ(\{q\})$ 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 $q_{\text{ref}}(o) \in Q$ be a query object such that $\forall q \in Q : d(o, q_{\text{ref}}(o)) \geq d(o, q)$. Then*

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

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

Proof Due to Lemma 1 we have

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

Again, let q_{ref} be the query point with the largest distance to o. Since $q_{\text{ref}} \in Q$ we get:

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

□

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, since, if the right-hand-side of 2 does not hold, the left-hand-side of Lemma 2 can not hold either.

5.1.3 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 $q_{\text{ref}}(o) \in Q$ a query object such that $\forall q \in Q : d(o, q_{\text{ref}}(o)) \geq d(o, q)$. Then for each object $p \in \mathcal{H}$ it holds that $d(o, p) \leq d(o, q_{\text{ref}}(o))$.*

Proof A formal proof of Lemma 3 can be found in Appendix. □

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 $q_{\text{ref}}(o) \in Q$ be a query object such that $\forall q \in Q : d(o, q_{\text{ref}}(o)) \geq d(o, q)$ for a given database object $o \in \mathcal{D}$. Then*

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

at most $k' = k - |\mathcal{H}| - |Q|$ objects $p \in \mathcal{D} - \mathcal{H}$ are closer to o than $q_{\text{ref}}(o)$, 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 $q_{\text{ref}}(o)$.

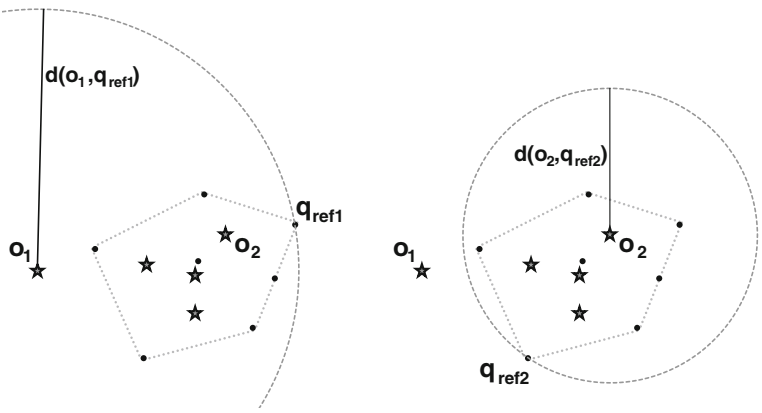
Proof

⇒ If $o \in Ik\text{-}NNQ(Q)$, then all query points (including q_{ref}) are in the k-NN set of o. Since for all points p in $\mathcal{H}$, $d(o, p) \leq d(o, q_{\text{ref}})$ (cf. Lemma 3), all points in $\mathcal{H}$ should also be in the k-NN set of o. Therefore, in the (worst) case, where q_{ref} is the k-th NN of o, there can be $k - |\mathcal{H}| - |Q|$ points outside the convex hull closer to o than q_{ref}, if $o \notin \mathcal{H}$, or $k - |\mathcal{H}| + 1 - |Q|$ points if $o \in \mathcal{H}$.

⇐ If o is outside the hull, from the points in $\mathcal{H} \cup Q$, q_{ref} is the farthest one to o (cf. Lemma 3). If there are at most $k - |\mathcal{H}| - |Q|$ points outside the hull closer to o than q_{ref} is, then the distance ranking of q_{ref} is at most k. Since all other points in Q are closer to o than q_{ref} is, it holds that $o \in Ik\text{-}NNQ(Q)$. If $o \in \mathcal{H}$, the bound is $k - |\mathcal{H}| - |Q| + 1$, as o should be excluded from $\mathcal{H}$ in the proof. □

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 $q_{\text{ref}}(o)$. 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 10-NN query using a set Q of seven query objects, shown as points in the figure; non-query database points are represented by stars. In Fig. 3a, the goal is to determine whether candidate object o_1 is a result,



(a) Pruning o_1. (b) Pruning o_2.

Fig. 3 Ik-NN pruning based on Lemma 4

i.e., whether o_1 has all $q \in Q$ in its 10-NN set. The query object having the largest distance to o_1 is $q_{\text{ref}}(o_1)$. 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 $q_{\text{ref}}(o_1)$. Thus, o_1 can be safely pruned without even considering these objects (since obviously, at least zero objects are closer to o_1 than $q_{\text{ref}}(o_1)$). Next, we consider object o_2 in Fig. 3b. The query object with the largest distance to o_2 is $q_{\text{ref}}(o_2)$. 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 $q_{\text{ref}}(o_2)$. This, 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 $q_{\text{ref}}(o_2)$.

5.1.4 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*, Papadias et al. [8]), which indexes $\mathcal{D}$, and computes *Ik-NNQ*(Q) for a given set Q of query objects, using Lemma 4 to prune the search space. The (*ARTree*) is a modified R-tree which stores in each node N the number of objects approximated by N. These counts can be used to accelerate search, as we will see later.

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 shows the 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 $q_{\text{ref}}(p)$ 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 the 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 subtrees 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,

Algorithm 1 Inverse k-NN query

Require: $Q, k, ARTree$
1: *//Fast Query Based Validation*
2: **if** $|Q| > k$ **then**
3: return no result and terminate algorithm
4: **end if**
5: pq : PriorityQueue ordered by $\max_{q_i \in Q}$MinDist
6: $pq.add(ARTree.root$ entries$)$
7: $|\mathcal{H}| = 0$
8: LIST *candidates*, *prunedEntries*
9: *//Query/Object Based Pruning*
10: **while** $\neg pq.isEmpty()$ **do**
11: $e = pq.poll()$
12: **if** $getPruneCount(e, Q, candidates, prunedEntries, pq) > k - |\mathcal{H}| - |Q|$ **then**
13: $prunedEntries.add(e)$
14: **else if** $e.isLeafEntry()$ **then**
15: $candidates.add(e)$
16: **else**
17: $pq.add(e.getChildren())$
18: **end if**
19: **if** $e \in convexHull(Q)$ **then**
20: $|\mathcal{H}| += e.agg\_count$
21: **end if**
22: **end while**
23: *//Refinement Step*
24: LIST *result*
25: **for** $c \in candidates$ **do**
26: **if** $q_{\text{ref}} \in knnQuery(c, k)$ **then**
27: $result.add(c)$
28: **end if**
29: **end for**
30: return *result*

and priority queue), we increase the *prune_counter* of e by the number of points in e' if the following condition holds: $\forall p \in e, \forall p' \in e' : dist(e, e') < dist(e, q_{\text{ref}}(p))$. This condition can efficiently be checked using the technique from [4]. An example were this condition is fulfilled is shown in Fig. 4. Here the *prune_counter* of e can be increased by the number of points in e'.

While updating the value of *prune_counter* for e, we perform a 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 Fig. 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.

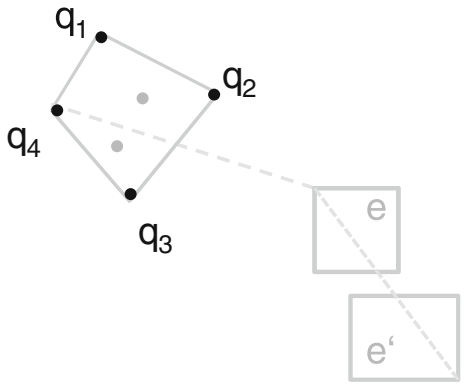


Fig. 4 Calculating the *prune_count* of *e*

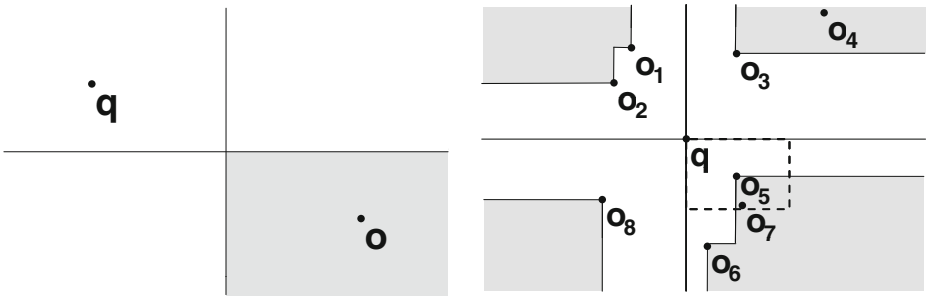
When the queue becomes empty, the filter step of the algorithm completes with a set of *candidates*. For each object c in this set, we check whether c is a result of the inverse query by performing a k-NN search and verifying whether its result includes Q. In our implementation, to make this test faster, we replace the k-NN search by an aggregate ε-range query around c, by setting $\varepsilon = d(c, q_{\text{ref}}(c))$. 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.

We further note, that although the bi-chromatic inverse k-NN case can be solved by simply disabling the object based filter, we note that bi-chromatic inverse k-NN queries are not very meaningful in practise. In this case, the task is to compute for a given database $\mathcal{D}$, the set of objects which have all query objects $q \in Q = \mathcal{D}'$ of a second database $\mathcal{D}'$ in their k-NN set. Clearly, this type of query returns nothing if $k < |Q|$, and the whole database $\mathcal{D}$ otherwise. This is evident, since for a bi-chromatic k-NN query, we drop the object-based filter, such that the query-based filter is required to drop k' to zero in order to produce results.

6 Inverse dynamic skyline query

Again, we 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. An 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 $q^i < o^i$ and $[-\infty, \frac{q^i+o^i}{2}]$ if



(a) Pruning region. (b) candidates.

Fig. 5 Single-query case

$q^i > o^i$. For $q^i = o^i$, the whole dimension is pruned, i.e. the pruning space is $[-\infty, \infty]$, except for the case where $q = o$, where there is no pruning space.[2]

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

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 Fig. 5b. The union of the four skylines is the set of the inverse query candidates (i.e. $\{o_1, o_2, o_3, o_5, o_6, o_8\}$ 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 $2 \cdot |c^i - q^i|$ in each dimension i is empty. For example, candidate o_5 in Fig. 5b 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.

[2]The later observation follows by the definition of domination, which allows to prune an object o' if there is another object o which is at least as close to o' as q in all dimension, and close in at least one dimension. In the case $o = q$, the case applies where distances in all dimensions are equal, so nothing can be pruned.

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

6.1 Framework implementation

6.1.1 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 an arbitrary 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.*

Proof Let us consider the space partitioning $\mathcal{S}$ derived from dividing the object space at q. Each q' located within partition $r \in \mathcal{S}$ generates a pruning region $PR_{q'}(q)$ (cf. Definition 2) that totally covers the partition $r' \in \mathcal{S}$ which is opposite to r w.r.t. q. Since we assume that we have at least one query object $q' \neq q$ in each partition $r \in \mathcal{S}$, all partitions $r' \in \mathcal{S}$ are totally covered by a pruning region and, consequently, the complete data space can be pruned. An example in the two-dimensional space is illustrated in Fig. 6. □

6.1.2 Query-based pruning

In the following, we show how pruning a data object from the candidates set of an inverse skyline query can be accelerated. Therefore, we first 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

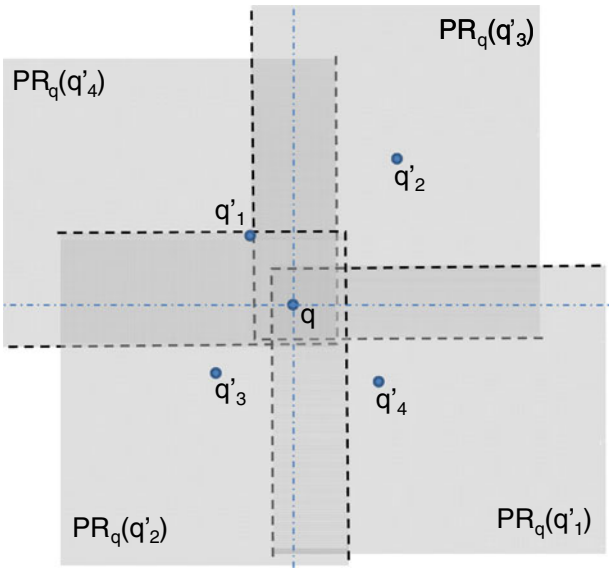


Fig. 6 Fast query based validation filter in 2D-space

Definition 2: $PR_q(q')$ and $PR_{q'}(q)$. Any object inside these regions cannot be a candidate for 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 use R-tree nodes that are contained in it. Figure 7 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.

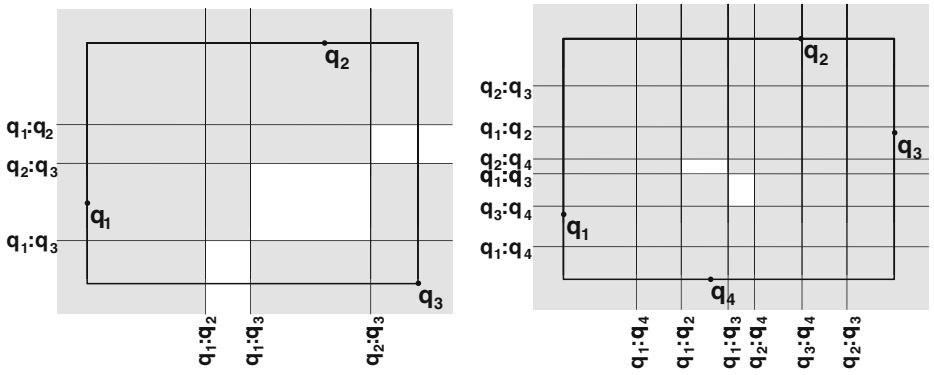
The main challenge is how to encode and use the pruning space defined by Q, as it can be arbitrarily complex in the multidimensional space. As for the *Ik-NNQ* case, our approach is not to explicitly compute and store the pruning space, but to check on-demand whether each object (or R-tree MBR) can be pruned by one or more query pairs. This has a complexity of $O(|Q|^2)$ checks per object.

Let $Q^{\square}$ be the rectangle that minimally bounds all query objects.

The main idea is to identify pruning regions by just considering $Q^{\square}$ and one query object q located on the boundary of $Q^{\square}$. We first concentrate on pruning objects outside of $Q^{\square}$, the cases for pruning objects inside of $Q^{\square}$ will be discussed later.

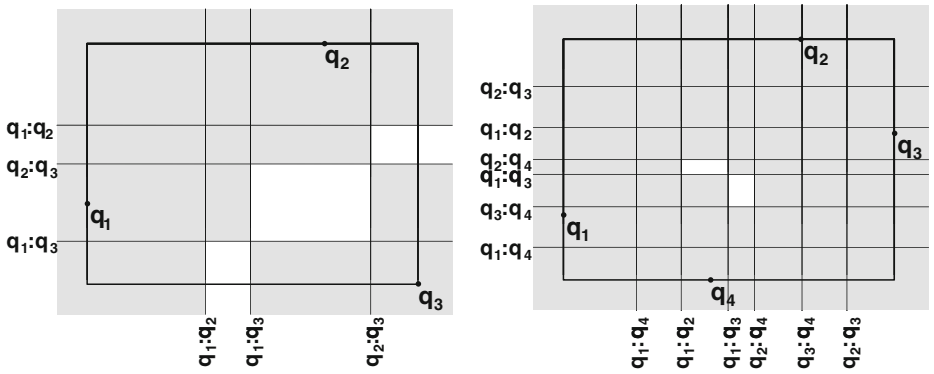
Pruning condition I Assume that q is located at one corner of $Q^{\square}$, then the axis-aligned region outside of $Q^{\square}$ where the ith dimension is defined by the interval $[c^i, +\infty]$ if $q^i > c^i$ and $[-\infty, c^i]$ otherwise (where c is the center of $Q^{\square}$) can be pruned. The rationale is that since $Q^{\square}$ is a *minimally* bounding rectangle, at least one other query object must be located at each edge of $Q^{\square}$ on which q is not located itself. By pairing q with each of these objects, and merging the corresponding pruning regions, we obtain the pruning region. In the example shown in Fig. 8a, q is located at the southeast corner of $Q^{\square}$ and, therefore, additional query objects must be located on both north and west edges of $Q^{\square}$. As a consequence, any object in the lower-right shaded region can be pruned.

Pruning condition II In the case, where q is located at a boundary of $Q^{\square}$, but not at a corner of $Q^{\square}$, the half-space constructed by splitting the data space along the face of $Q^{\square}$ containing q and not containing $Q^{\square}$ defines the pruning region. Here, the rationale is that since q is on an face f (but not at a corner) of $Q^{\square}$, there must



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

Fig. 7 Pruning regions of query objects



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

Fig. 8 Using query points at the border of $Q^{\square}$ to prune space

be at least $2 \cdot (d-1)$ additional query objects, located at the edges adjacent to f. By pairing q with each of these query objects, and merging the corresponding pruning regions, we obtain the final pruning region. For example, in Fig. 8b, q is paired with two more query objects located on the east and the west edge of $Q^{\square}$. We obtain the shaded pruning region below q. As another example, consider the union of $PQ_{q_1}(q_4)$ and $PQ_{q_3}(q_4)$ in Fig. 7b which prunes the whole hyperplane below q_4. Thus, if all four edges of $Q^{\square}$ contain four different query objects, then only objects in $Q^{\square}$ are candidates for *I-DSQ* results.

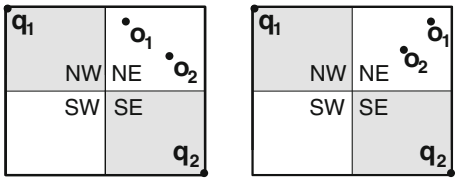
6.1.3 Object-based pruning

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. Naively, we can determine, for each database object o' that we have found so far and each query object q, the pruning region $PR_q(o')$ according to Definition 2, and subsequently check whether o is located in this region. In the following, we show how to perform this pruning without considering all possible combinations of database and query objects.

Consider two query points q_1 and q_2 and the rectangle $Q^{\square}_{q_1q_2}$ that minimally bounds q_1 and q_2. Note, that $Q^{\square}_{q_1q_2}$ is fully contained in $Q^{\square}$ and for each pair $q_i, q_j \in Q, i \neq j$ the union of all $Q^{\square}_{q_iq_j}$ is equal to $Q^{\square}$. Consider the axis-aligned space partitioning according to the center point c of $Q^{\square}_{q_1q_2}$ resulting in 2^{d-1} partitions, denoted as NE, SE, SW and NW in Fig. 9. According to pruning condition I, the two partitions containing q_1 and q_2 respectively, can be pruned.

For the two-dimensional case, this observation leads to a powerful pruning criterion (pruning criterion III) which uses the following corollary:

Corollary 1 *Let $Q^{\square}_{q_1q_2}$ be a minimal rectangular region minimally bounding two query objects q_1 and q_2. Consider the two regions R_1 and R_2 given by the axis-aligned*



(a) Case 1. (b) Case 2.

Fig. 9 Object-based pruning inside $Q^{\square}$

space partitioning according to the center point c of $Q^{\square}_{q_1 q_2}$ that cannot be pruned using pruning criterion I. In the two-dimensional case, each of these regions may contain at most one candidate object for a 1-DSQ query.

Proof To prove this, let us consider the following two cases illustrated in Fig. 9a and b:

Case 1 Let $SR(o_i, o_j) \in \{NW\text{-}SE, NE\text{-}SW\}$ denote the *spatial relationship* between two objects o_i and o_j. We define

$$SR(o_i, o_j) = \begin{cases} NE\text{-}SW, & \text{if } o_i.x > o_j.x \wedge o_i.y > o_j.y \vee o_i.x < o_j.x \wedge o_i.y < o_j.y \\ NW\text{-}SE & \text{otherwise} \end{cases},$$

where $o.x$ and $o.y$ denote the value of the first and second dimension of object o, respectively.

Let us assume that there are two objects o_1 and o_2 in one of the regions $R \in \{R_1, R_2\}$, that have the same spatial relationship as q_1 and q_2, i.e. $SR(o_1, o_2) = SR(q_1, q_2)$, as shown in Fig. 9a. In this case, both objects o_1 and o_2 must prune each other. The reason is that o_1 is in the pruning region of o_2 w.r.t. q_2 and o_2 is in the pruning region of o_1 w.r.t. q_1. Consequently, for any object o_i in R to be a candidate, there cannot exist another object o_j such that $SR(o_i, o_j) = SR(q_1, q_2)$.

Case 2 Now, we assume that there are two objects o_1 and o_2 in one of the regions $R \in \{R_1, R_2\}$, that do not have the same spatial relationship as q_1 and q_2, i.e. $SR(o_1, o_2) \neq SR(q_1, q_2)$, as shown in Fig. 9b. In this case, it must hold that either o_i is pruned by o_j or vice versa. The reason is that in each dimension, and for both $q_i \in \{q_1, q_2\}$, it holds that the distance between o_i and o_j must be less than the distance between o_i and q_i, or vice versa. For example, assume that R corresponds to the north-east region of center c, as in Fig. 9b. Here, o_2 can be pruned by o_1. However, merely based on the spatial relation between o_2 and o_1, we can not decide whether o_1 can be pruned by o_2. In this example, this is not the case, but would be the case if o_2 was closer to o_1.

Clearly there can be at most one object $o_i \in R$ such that for each other object o_j in R, $o_i \neq o_j$, both query objects $q_i \in \{q_1, q_2\}$ and each dimension it holds that that the distance between o_i and o_j must be less than the distance between o_i and q_i. This is the object closest to the corner of $Q^{\square}$ contained in R. □

We can now use Corollary 1 to obtain the following pruning condition:

Pruning condition III Let $R \subseteq Q^{\square}$ be a region inside the query rectangle that cannot be pruned using query-based pruning. Let $q_1, q_2 \in Q$ be two query points for which R is fully contained in the rectangle $Q^{\square}_{q_1 q_2}$ minimally bounding q_1 and q_2. Since R cannot be pruned based on query-pruning only, R must be located in non-pruning regions (e.g. NE and SW in Fig. 7a) of q_1 and q_2. Without loss of generality, let us assume that R is located in the NE region of $Q^{\square}_{q_1 q_2}$. Now let O be the set of database objects inside R. Let $a \in O$ be the object with the largest x coordinate and let $b \in O$ be the object with the largest y coordinate. If $a \neq b$ we can prune R. If $a = b$, then a is a candidate and all other objects $c \in R, c \neq a$ can be pruned.

The above pruning condition allows us to prune objects inside $Q^{\square}$ for the two-dimensional case. The next pruning condition allows us to prune objects outside of $Q^{\square}$ using database objects for arbitrary dimensionality. In the following, let $Q^{\square}_i.\max$ and $Q^{\square}_i.\min$ denote the maximum and minimum coordinate of $Q^{\square}$, at dimension i, respectively.

Pruning condition IV For the next pruning condition we use the sets of database objects $O_i \subseteq \mathcal{D}$ outside $Q^{\square}$ for which it holds that each $o \in O_i$ intersects $Q^{\square}$ in all but one dimension. Such objects are shown in Fig. 10 for the two-dimensional case. Let O_i^+ (O_i^-) denote the subset of O_i such that each object $o \in O_i^+$ has a larger (smaller) coordinate than $Q^{\square}$ in the non-intersecting dimension. Now let $o_i^+ \in O_i^+$ ($o_i^- \in O_i^-$) be the object in O_i^+ (O_i^-) with the smallest (largest) coordinate in the non-intersecting dimension j. Any database object which has a j coordinate greater than $\frac{o_i^+ + Q_i^{\square}.\max}{2}$ or less than $\frac{o_i^- + Q_i^{\square}.\min}{2}$ can be pruned. The rationale of this pruning condition is that due the MBR property of $Q^{\square}$, we know that at least one query object must be located on each edge of $Q^{\square}$. The pruning regions defined are based on these query objects. For instance, for object o_1^+ in Fig. 10 we can exploit that there must be query objects q_1 and q_2 located on the left and the right border edge of $Q^{\square}$, respectively. This allows us to create the two pruning regions $PR_{q_1}(o_1^+)$ and $PR_{q_2}(o_1^+)$ according to Definition 2. These pruning regions are smallest possible, if q_1 and q_2 are located at the upper corners of $Q^{\square}$. Thus, we can prune any object above the line bisecting the upper side of $Q^{\square}$ and o_1^+.

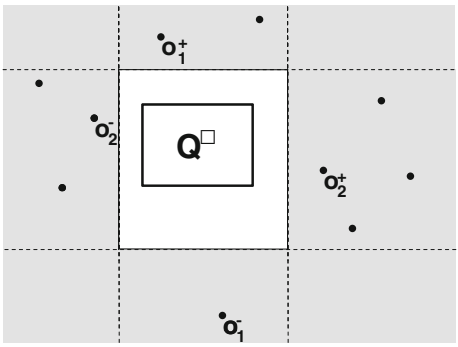


Fig. 10 Pruning regions outside of $Q^{\square}$

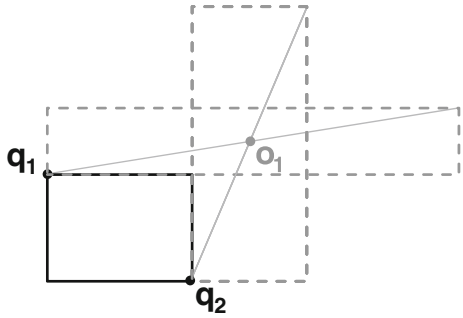


Fig. 11 Refinement area defined by q_1, q_2 and o_1

6.1.4 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. Fig. 11): 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.

Algorithm 2 Inverse dynamic skyline query

Require: Q, $ARTree$

```
1:  pq : PriorityQueue ordered by min_{q_i∈Q} MaxDist
2:  pq.add(ARTree.root entries)
3:  LIST candidates, prunedEntries
4:  //Filter step
5:  while ¬pq.isEmpty() do
6:     e = pq.poll()
7:     if canBePruned(e, Q, candidates, prunedEntries, pq) then
8:        prunedEntries.add(e)
9:     else if e.isLeafEntry() then
10:       candidates.add(e)
11:    else
12:       pq.add(e.getChildren())
13:    end if
14: end while
15: //Refinement Step
16: LIST result
17: for c ∈ candidates do
18:    if Q ∈ dynamicSkyline(c) then
19:       result.add(c)
20:    end if
21: end for
22: return result
```

6.2 Algorithm

The pseudocode for *I-DSQ* is shown as Algorithm 2. During the filter steps, the tree is traversed in a best-first manner, where entries are accessed by their minimum 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-NN* case, lists for the candidates and pruned entries are maintained. The pruning conditions of Section 6.1 are used wherever applicable to reduce the computational cost. Finally, the remaining candidates are refined using the refinement strategy described in Section 6.1.

For the case of bi-chromatic inverse dynamic skyline queries, the pruning region is only defined by objects in the set of query objects $q \in Q = \mathcal{D}$. This pruning region is used to prune objects o in $\mathcal{D}'$. Again, the object-based filter is dropped, since in the bi-chromatic case, objects in the set $\mathcal{D}'$ do not affect each other.

7 Relaxed inverse queries

A major disadvantage of inverse queries in general is the problem that often no result is returned. This problem becomes evident by considering the fast query-based evaluation steps for each query predicate: An inverse query will return no result at all, if a simple condition is satisfied. In many applications, this becomes a very constraining concern: For example, in the family movie application there may be no single movie which is sufficiently similar to the preference of all family members, thus an inverse query would return an empty set. Instead, it may be preferred to return movies which are sufficiently similar to as many family members as possible.

In this section, we will investigate an approach to ensure that a meaningful result is returned. Therefore, we will relax Definition 1 by no longer requiring that, for a database object o to be a result of an inverse query, all query objects Q have to be in the result the $\mathcal{P}$ query result of o. Instead, we want to return the set of objects having the largest number of objects $q \in Q$ in their $\mathcal{P}$ query result. Thus, if there is no object which qualifies for the inverse query predicate for *all* objects in Q, we will return all objects that qualify for exactly $|Q| - 1$ objects in Q. If there is no such object, we will return all objects that qualify for exactly $|Q| - 2$ objects in Q, and so on. We redefine inverse queries as follows:

Definition 3 A relaxed inverse $\mathcal{P}$ query ($I_{\text{relax}}\mathcal{P}$ query) computes for a given set of query objects $Q \subseteq \mathcal{D}$ the set of points $r \in \mathbb{R}^d$ for which the largest number of objects $q \in Q$ are in the $\mathcal{P}$ query result; formally:

$$I_{\text{relax}}\mathcal{P}(Q) = \{r : |\mathcal{P}(r)| = \max_{r\in R^d}(|\mathcal{P}(r) \cap Q|)\}$$

In the following, we will discuss solutions for $I_{\text{relax}}\mathcal{P}$ queries for the mono-chromatic case. Note that Definition 3 is guaranteed to yield at least one answer, because in the worst case it degenerates to a reverse query (for any $q \in Q$).

A naive approach to compute the result of an $I_{\text{relax}}\mathcal{P}$ query is to iteratively consider the set $\mathcal{S}_k$ of $\binom{|Q|}{k}$ subsets of Q of size k, starting at $k = |Q|$ and decrementing

k in each iteration. For each $s \in \mathcal{S}_k$, an $I\mathcal{P}(s)$ query is performed. An iteration terminates when a non-empty result is returned, and all results of the same iteration are returned. Clearly, this approach is not viable for large query sets Q due to exponential runtime in $|Q|$.

Another naive approach performs a reverse $\mathcal{P}$ query for each query object $q \in Q$. Then, for each database object o we count the number of occurrences of o in the results of the reverse queries and return the set of objects with the largest number of occurrences.

We propose to extend the latter approach by performing all reverse queries in a parallel fashion, i.e. in one single traversal of the R*-tree. Depending on the respective query predicate $\mathcal{P}$, we can identify pairs of query objects q and entries e of the R*-tree (which can be individual database objects or intermediate index pages), for which we can already make a decision (positive or negative) whether e must have q in its $\mathcal{P}$ query result. These decisions, allow us to estimate the number of query objects in the $\mathcal{P}$ query result of e. Let $LB(e)$ and $UB(e)$ be lower and upper bounds of the number of query objects in the result of e, respectively. Using these bounds, we can derive a lower bound $maxMin$ of the highest number of query points that any database object has in its $\mathcal{P}$ query result. In a nutshell, $maxMin$ is the highest lower bound $LB(e)$ of all active entries e of the R*-tree at a given time:

$$maxMin = \max_e LB(e)$$

The bound $maxMin$ allows to prune entries e which cannot possibly have $maxMin$ or more query objects in their result set, i.e. objects where $UB(e) < maxMin$.

An example is given in Fig. 12, where a set of three query points $Q = \{q_1, q_2, q_3\}$ and a set of intermediate R*-tree entries are depicted. Here, we assume that the predicate $\mathcal{P}$ is given as ε-range with the depicted radius as parameter. For each entry $e \in \{e_1, ..., e_5\}$, the number of query objects in the range of the approximated database objects can be approximated easily: The minimum number is given by the ranges which fully contain e, whereas the maximum number is given by the ranges intersecting e. For instance, each database object approximated by entry e_1 must have exactly one query object in its ε-range, since q_3 must be in the range, while q_1 and q_2 cannot possibly be in the range. Entry e_2 has between zero and one objects in its range, because q_1 may or may not be in the range of children of e_2. Entry e_4 has a minimum number of two query objects in the range of its children, which is the highest lower bound of all entries, thus $maxMin = 2$. This allows us to prune all entries having an upper bound of less than two query objects, i.e. entries e_1,e_2 and

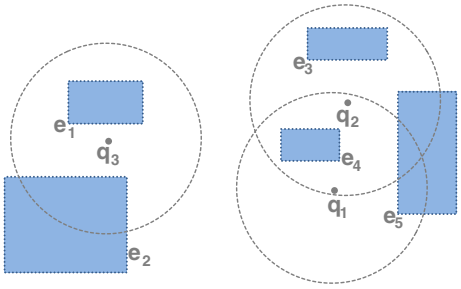


Fig. 12 Example for a $I_{\text{relax}}\mathcal{P}$ query

e_3 can be safely pruned. Children of e_5 however may also have two query objects in their range, and thus, have to be considered, therefore e_5 has to be refined.

A formalization of this approach for an arbitrary predicate $\mathcal{P}$ is shown in Algorithm 3. This algorithm requires a set of query objects Q, a database indexed by an R*-tree $RTree$ and a query predicate $\mathcal{P}$ with corresponding parameters (e.g.

Algorithm 3 Relaxed Inverse $\mathcal{P}$ query

```
Require: Q, P, RTree
pq : PriorityQueue ordered by UB_P.
pq.add(RTree.root entries)
maxMiPruneCount = 0
LIST candidates, prunedEntries
//Query/Object Based Pruning
while ¬pq.isEmpty() do
    e = pq.poll()
    if (LB_P(e) > maxMin) then
        maxMin = LB_P(e)
    end if
    if UB_P(e) < maxMin then
        prunedEntries.add(e)
    else if e.isLeafEntry() then
        candidates.add(e)
    else
        for (childine.getChildren()) do
            if (LB_P(child) > maxMin then
                maxMin = LB_P(child)
            end if
            if UB_P(child) < maxMin then
                prunedEntries.add(child)
            else
                pq.add(child)
            end if
        end for
    end if
end while
//Refinement Step
LIST result
maxResults = 0
for c ∈ candidates do
    if |PQuery(c) ∩ Q| > maxResults then
        result.clear()
        result.add(c)
    else if |PQuery(c) ∩ Q| = maxResults then
        result.add(c)
    end if
end for
return result
```

k for k-NN queries, $\varepsilon ilon$ for $\varepsilon ilon$-range queries). Similar to our algorithms for $I\mathcal{P}$-queries, we propose to use a best-first approach to traverse the tree. Thus, all current entries are stored in a priority queue, using meaningful heuristics for the sorting, such as average minimum distance to all query objects.

In each iteration, the first object e of the queue is dequeued and the number of query objects in its $\mathcal{P}$ query result is lower- and upper-bounded using the functions $LB(e)$ and $UB(e)$. Obtaining these approximations depends on the predicate $\mathcal{P}$:

- For $\mathcal{P} = \varepsilon$-range, a lower bound is given by the number of ε-range spheres completely containing e, whereas an upper bound is given by the ε-range spheres intersecting e.
- For $\mathcal{P} = k$-NN, we propose to use the concept of spatial domination [4], to decide, for each $q \in Q$, if q must be (must not be, may be) a k-NN of e. A lower bound is then given by the total number of query points that must be k-NNs of e, while an upper bound is given by the number of query objects that must or may be k-NNs of e.
- Analogously, for $\mathcal{P} = DS$, the spatial pruning techniques proposed in [7] can be used to find query objects $q \in Q$ that must (not) be in the dynamic skyline of e.

If the upper bound of e is less than the current maximum lower bound $maxMin$, then we can conclude that there must be an object in the database that has more query objects in its result than any object in e, and thus e can be pruned. If the lower bound of e is greater than the current maximum lower bound $maxMin$, then $maxMin$ is adjusted accordingly. In this case, we can prune all objects (both in the candidate list *candidates* and in the priority queue pq which have an upper bound less than the new $maxMin$). Then, if e is a leaf-node and has not been pruned, it is added to the candidate list. Otherwise, if e is an intermediate entry, it is being resolved, and its children are checked whether they can be pruned or whether they can increase $maxMin$. The pruning phase stops when the priority queue pq is empty, which implies that there is no intermediate entry left that can possibly be pruned. At this point, the set of candidate objects is refined, i.e. their $\mathcal{P}$-query result is computed naively, and all objects having the largest number of query objects in their result are returned.

8 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**). **Naive** computes the corresponding reverse queries for every $q \in Q$ and intersects their results iteratively. In order to be fair, we terminated **Naive** as soon as the intersection of results obtained so far is empty. **SQF** performs an 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 1 kB to index the datasets 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 in the charts. In all settings, we performed 1,000 queries and averaged the results. All methods were implemented in Java 1.6; tests were run on a dual core (3.0 GHz) workstation with 2 GB main memory having Windows XP as OS. We note that all our experiments were performed in main memory, however, to give an intuition about the additional cost for hard-disc access, we further evaluated the number of page-accesses that would be required using a disc-based R*-tree. The performance evaluation settings are summarized below; the numbers in **bold** correspond to the default settings:

Parameter	Values
db size	100,000 (synthetic), 175,812 (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 datasets:

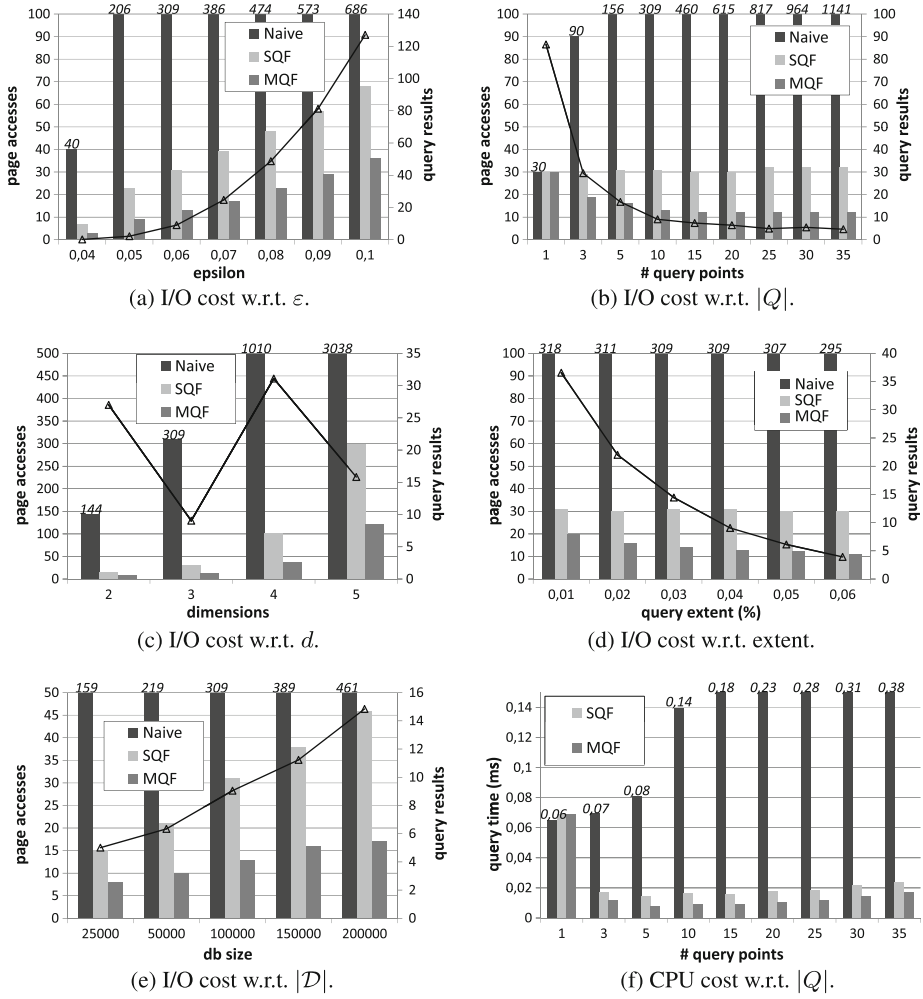
- Synthetic datasets: Clustered and uniformly distributed objects in d-dimensional space.
- Real dataset: Vertices in the Road Network of North America,[4] contains 175,812 two-dimensional points.

The datasets 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 maximum 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 yielding more results. In this section, we show the behavior of all three algorithms on the uniform datasets only. Finally, the evaluation of the relaxed version of inverse queries are presented in Section 8.4.

8.1 Inverse ε-range queries

We first evaluated the algorithms on inverse ε range queries. Figure 13a 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 $O(\varepsilon^d)$. Note that due

[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. 13 $I\varepsilon$-Q algorithms on uniform dataset

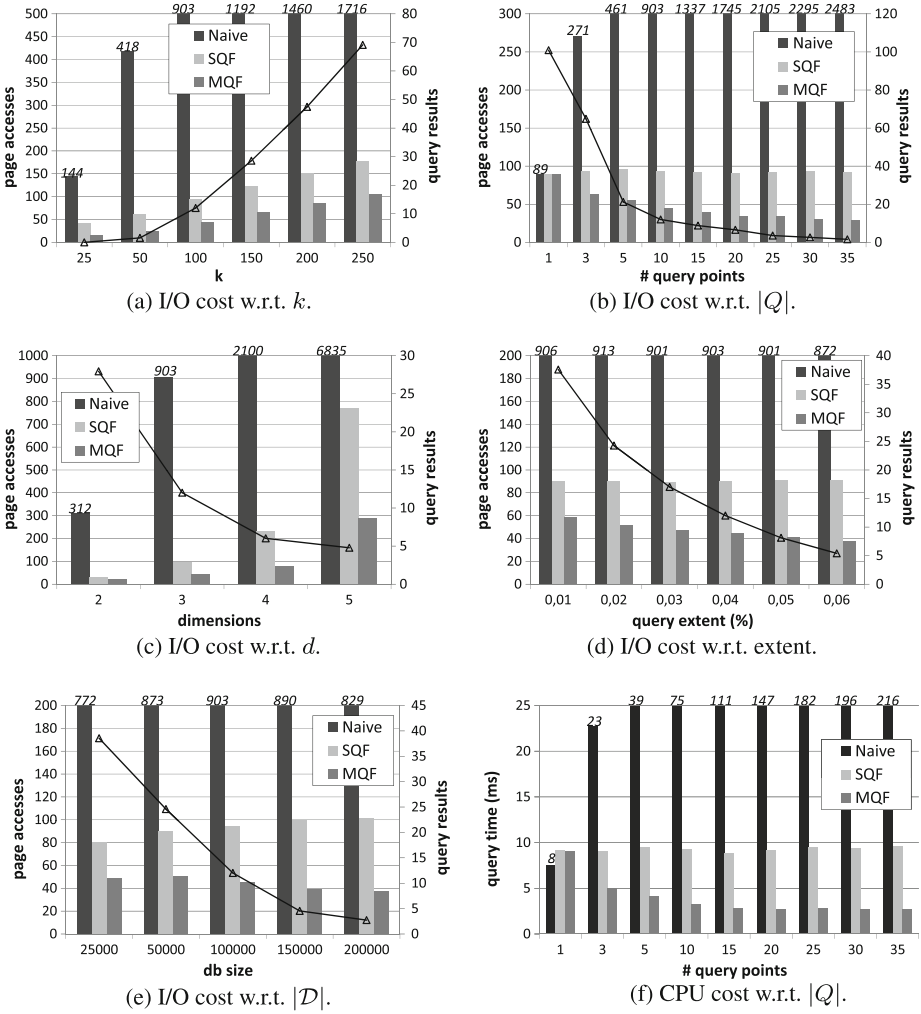
to the high run-time and the large number of page accesses of **Naive**, we further annotated the measured values above the corresponding bars.

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 Fig. 13b), as in this case there are more intersecting hyperspheres. The I/O cost of **SQF** in this case remains almost constant which is mainly due to the use of the buffer lowering the page accesses in the refinement step. Similar results can be observed when varying the database size (cf. Fig. 13e) and query extent (cf. Fig. 13d). For the data dimensionality experiment (cf. Fig. 13c) we set epsilon such that the sphere defined by ε covers always the same percentage of the data space, 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 **MQF** copes better with data dimensionality than the other approaches. In a last experiment (cf. Fig. 13f) we compared the computational costs of the algorithms. Even though Inverse Queries are I/O bound, **MQF** is still preferable for main-memory problems.

8.2 Inverse k-NN queries

The three approaches for inverse k-NN search show a similar behavior as those for the $I\varepsilon$-RQ. Specifically the behavior for varying k (cf. Fig. 14a) is comparable to varying ε and increasing the query number (cf. Fig. 14b) and the query extent (cf. Fig. 14c) yields the expected results. When testing on datasets with different dimensionality, the advantage of **MQF** becomes even more significant when d

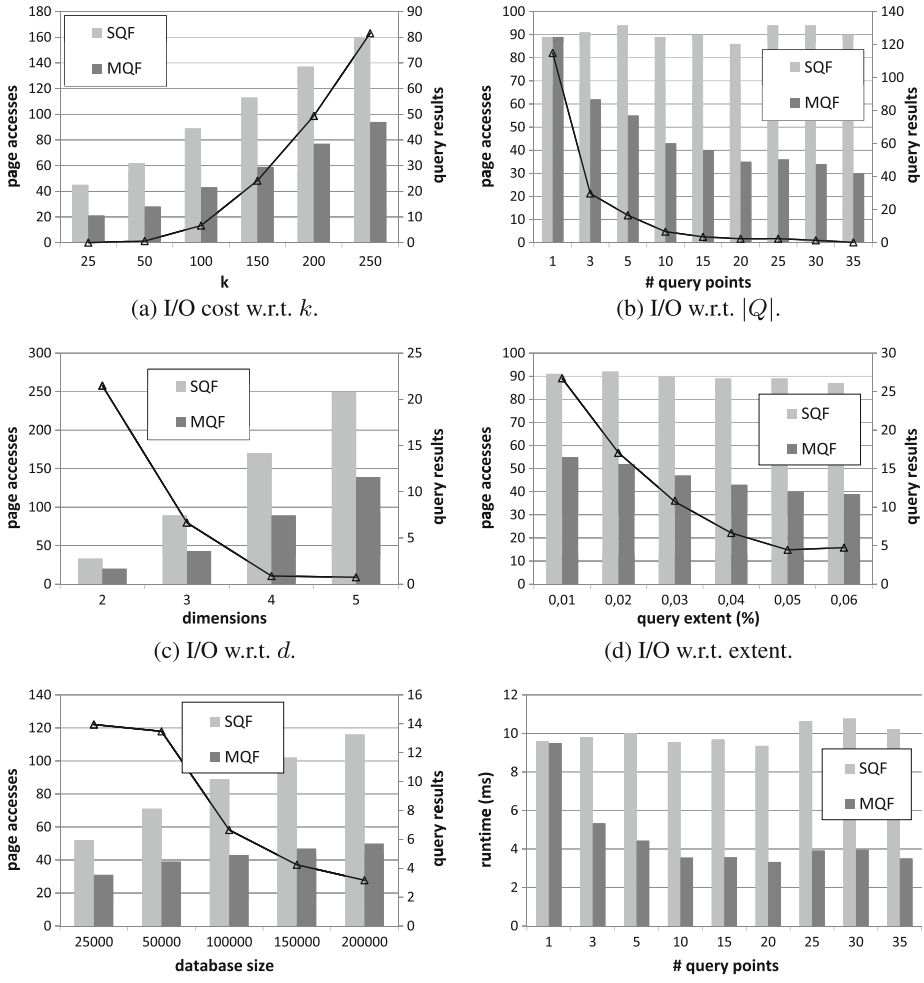


(a) I/O cost w.r.t. k.
(b) I/O cost w.r.t. $|Q|$.
(c) I/O cost w.r.t. d.
(d) I/O cost w.r.t. extent.
(e) I/O cost w.r.t. $|\mathcal{D}|$.
(f) CPU cost w.r.t. $|Q|$.

Fig. 14 *Ik-NNQ* algorithms on uniform dataset

increases (cf. Fig. 14d). In contrast to the $I\varepsilon$-RQ results for Ik-NN queries the page accesses of **MQF** decrease (cf. Fig. 14e) 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 $I\varepsilon$-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 (cf. Fig. 14f).

For further evaluation of Ik-NN queries on the synthetic clustered dataset (cf. Fig. 15) and the real dataset (cf. Fig. 16), we excluded from the evaluation the naive approach due to its poor performance; this way, the difference between **MQF** and



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

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

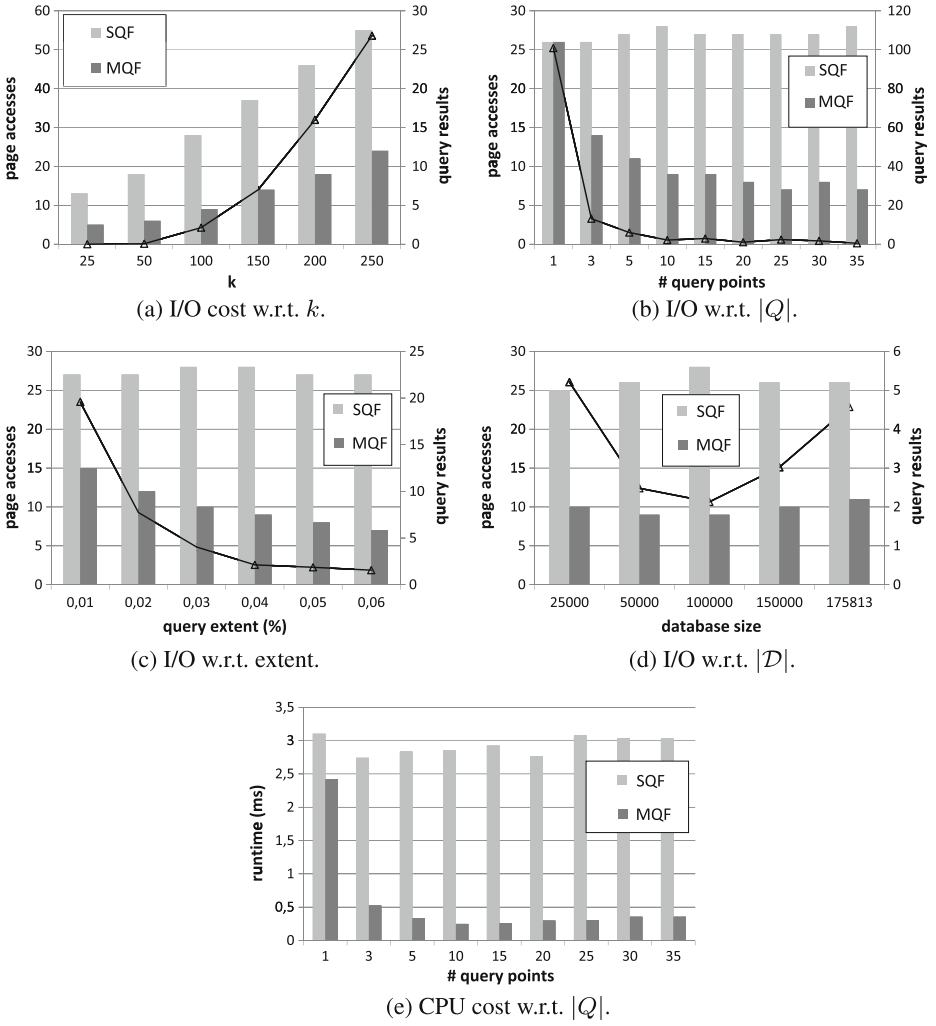
(c) I/O w.r.t. d.

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

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

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

Fig. 15 *Ik-NNQ* algorithms on clustered dataset



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

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

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

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

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

Fig. 16 *Ik-NNQ* algorithms on real dataset

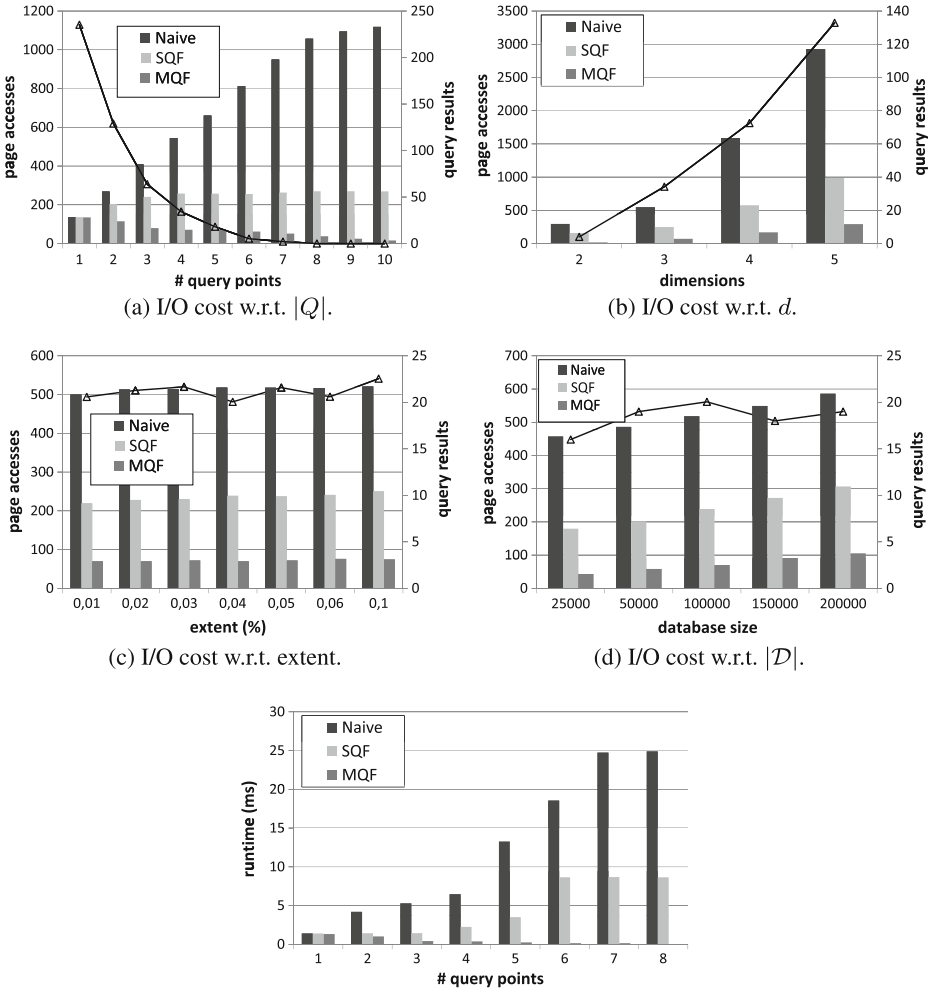
SQF becomes more clear. Let us note that similar trends could be observed for the $I\varepsilon$-range query.

8.3 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 dataset (cf. Fig. 17b), thus this value can be much larger for higher dimensional datasets.

Increasing the distance among the queries does not affect the performance as seen in Fig. 17c; regarding the number of results in contrast to inverse range- and k-NN-queries, inverse dynamic skyline queries are almost not sensitive to the distance among the query points. The rationale is that dynamic skyline queries can have results which are arbitrarily far away from the query point, thus the same holds for the inverse case. The same effect can be seen for increasing database size (cf. Fig. 17d). The advantage of **MQF** remains constant over the other two approaches. Like inverse range- and k-NN-queries, *I-DSQ* are I/O bound (cf. Fig. 17e), but **MQF** is still preferable for main-memory problems.

Figure 18 presents the results obtained with experiments on the synthetic clustered dataset, where similar results were obtained.



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

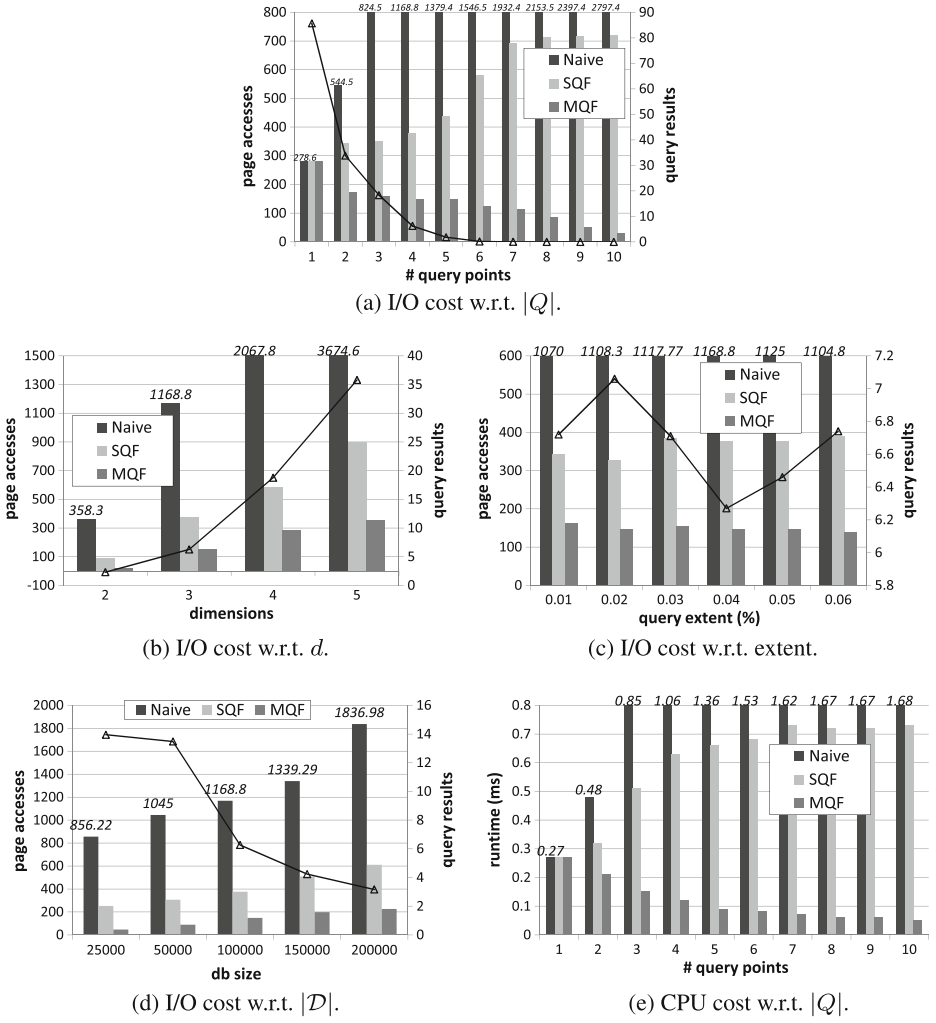
(b) I/O cost w.r.t. d.

(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. 17 *I-DSQ* algorithms on uniform dataset



(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. $|\mathcal{D}|$.

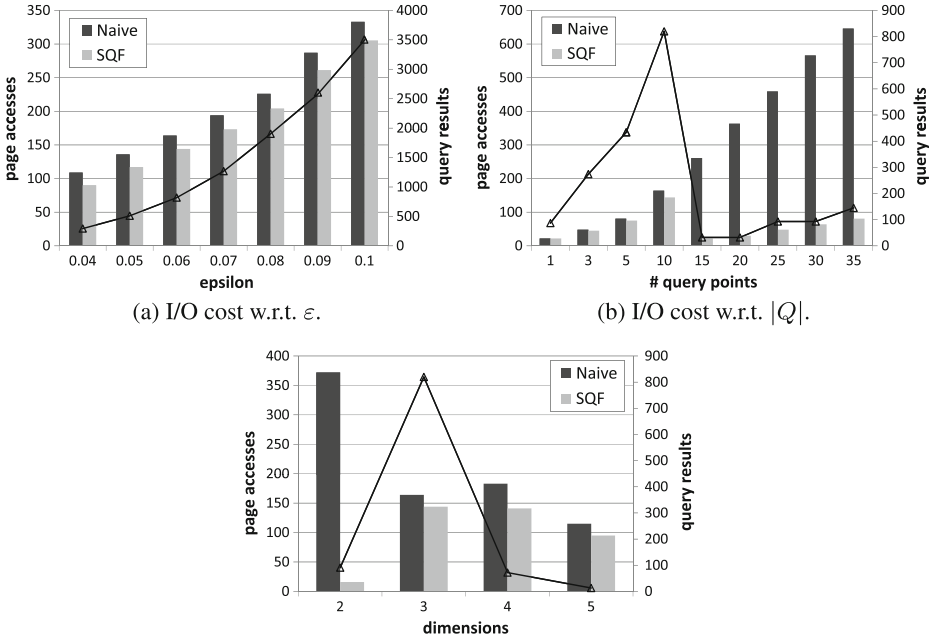
(e) CPU cost w.r.t. $|Q|$.

Fig. 18 *I-DSQ* algorithms on clustered dataset

8.4 Relaxed inverse ε-range queries

In this section, we review the evaluation of relaxed inverse queries $I_{\text{relax}}\mathcal{P}$ that have been introduced in Section 7. In this case, the query predicate $\mathcal{P}$ was evaluated using ε-range. We compare the approach based on single-query-filtering (**SQF**) with a naive approach (**Naive**) on the uniformly distributed dataset. Note that, for this dataset, there is little overlap of the ε spheres for $d \geq 3$. Thus, the maximum number of query objects is 1 for $d \geq 3$ in most of the 1,000 runs. The results are summarized in Fig. 19. Here again, the number of page accesses as well as the number of query results are of interest.

Varying the range value ε for $I_{\text{relax}}\varepsilon$-$Q$ algorithms, we obtain the results depicted in Fig. 19a. It is obvious that, with increasing ε, the number of results increases as



(a) I/O cost w.r.t. ε.

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

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

Fig. 19 $I_{\text{relax}}\varepsilon$-$Q$ algorithms on uniform dataset

well, as, for each query, more objects are likely to be included in the ε spheres. For both approaches, the number of page accesses also increases, since more tree entries intersect the spheres when the radius of the spheres is large enough.

Next, we evaluated the number of query points in Q. As Fig. 19b shows, there is a significant break point in the step from 10 to 15 query objects regarding the number of page accesses of **SQF** as well as the number of results. Up to 10 query objects, these values increase linearly. This is due to the fact that the number of covered data points by the union of all ε spheres increases constantly for each query, if the objects uniformly distributed in the data space. For ≤ 10 query objects, the ε spheres of the queries do not overlap at all, so each of the returned results qualify for exactly one object in Q. A query set size of greater then 10 leads to increasing overlap between two ε spheres, respectively. These overlapping regions contain the results of the next stage – the results obtained now qualify for exactly two objects in Q. However, a large number of objects do not appear in the result set now, as they only have one query object in their ε sphere. As **Naive** first computes the results for each query and applies the intersections afterwards, the number of page accesses further increases linearly. Applying the pruning rules of Section 7, **SQF** ia able to discard the objects that do not qualify to be part of $I_{\text{relax}}\varepsilon$-$Q$. Therefore, the number of accessed pages with **SQF** starts to be low again at this break point.

The results with varying dimensionality of the data are depicted in Fig. 19c. Regarding the number of results, there is again a significant break point, in this case for the steps from two to three and from three to four dimensions. For the three-dimensional case, there is hardly any overlap of the ε spheres. Thus, similar to the

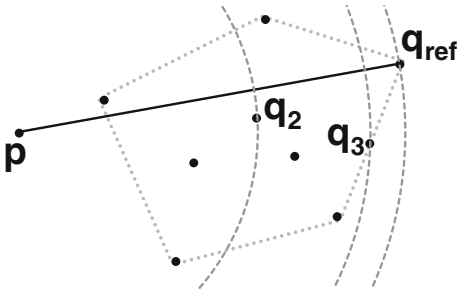


Fig. 20 Illustration of Corollary 2

case having 10 query points above, the result objects qualify for exactly one object in Q. Having $d = 2$, the pruning conditions of **SQF** work well due to overlapping ε spheres. With $d = 3$, the number of pages which are intersected by an ε sphere and thus have to be considered is significantly higher. **Naive** produces a high number of page accesses with $d = 2$, as the ε spheres cover large parts of the data space. The step to $d = 3$ is again significant due to the shrinking coverage of the spheres of the data space. With increasing dimensionality ($d \geq 3$), both algorithms perform similarly w.r.t. the number of accessed pages. This number then decreases having $d = 5$, since the branching factor of the R*-tree increases.

9 Conclusions

In this paper we introduced and formalized the problem for inverse query processing. We proposed a general framework for 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. Furthermore, we relaxed the definition of inverse queries in order to ensure non-empty result sets, which is vital for many applications. 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. 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 German Academic Exchange Service (Proj. ID 50149322).

Appendix: Proof of Lemma 3

We first require the following corollary:

Corollary 2 *Let $Q \in \mathbb{R}^d$ be a set of points and $C \subseteq Q$ be the vertices of the convex hull of Q in $\mathbb{R}^d$. Then, for each point $p \in \mathbb{R}^d$, the farthest point in Q to p must be in C as well.*

Proof Consider any point $p \in \mathbb{R}^d$ and its farthest point $q \in Q$. Then all points in Q must be located in the hyper-sphere centered at o with radius $d(o, q)$. Now, we can proof the above lemma by contradiction assuming that q is not a convex-hull vertex. If q is assumed to be within the convex-hull (not lying on the margin of the convex hull), then the hyper-sphere splits the convex-hull into points that are inside the sphere and points that are out-side of the sphere as shown for q_2 in Fig. 20. Consequently, the convex hull contains points that are farther from o than q which contradicts the assumption. Now, we assume that q lies on the margin (but not on a vertex) of the convex hull which corresponds to a region of a hyper-plane like q_3 in our example. If we move along this hyper-plane starting from q, we are still within the convex-hull but leave the hyper-sphere of o. Consequently, again, the convex hull contains points that are farther from o than q which again contradicts the assumption. □

Now we can use Corollary 2 to prove Lemma 3:

Proof By definition of q_{ref} it holds that

$$q_{\text{ref}} = \text{argmax}_{q \in Q}(dist(o, q))$$

Since the vertices C of the convex hull of Q consists only of points in Q, Corollary 2 leads to

$$q_{\text{ref}} = \text{argmax}_{c \in C}(dist(o, c))$$

Thus,

$$\forall c \in C : dist(o, q_{\text{ref}}) \geq dist(o, c)$$

and since $\mathcal{H} \subseteq C$:

$$\forall p \in \mathcal{H} : dist(o, q_{\text{ref}}) \geq dist(o, p).$$

□

References

1. Beckmann N, Kriegel H-P, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proc. SIGMOD
2. Chen L, Lian X (2008) Dynamic skyline queries in metric spaces. In: EDBT, pp 333–343
3. Dellis E, Seeger B (2007) Efficient computation of reverse skyline queries. In: VLDB, pp 291–302
4. Emrich T, Kriegel H-P, Kröger P, Renz M, Züfle A (2010) Boosting spatial pruning: on optimal pruning of mbrs. In: SIGMOD, 6–11 June 2010
5. Hjaltason GR, Samet H (1995) Ranking in spatial databases. In: Proc. SSD
6. Korn F, Muthukrishnan S (2000) Influence sets based on reverse nearest neighbor queries. In: Proc. SIGMOD
7. Lian X, Chen L (2008) Monochromatic and bichromatic reverse skyline search over uncertain databases. In: SIGMOD conference, pp 213–226
8. Papadias D, Kalnis P, Zhang J, Tao Y (2001) Efficient olap operations in spatial data warehouses. In: Proc. SSTD
9. Papadias D, Shen Q, Tao Y, Mouratidis K (2004) Group nearest neighbor queries. In: Proceedings of the 20th international conference on data engineering, ICDE 2004, 30 March–2 April 2004. Boston, MA, pp 301–312

10. Papadias D, Tao Y, Mouratidis K, Hui CK (2005) Aggregate nearest neighbor queries in spatial databases. ACM Trans Database Syst 30(2):529–576
11. Sharifzadeh M, Shahabi C (2006) The spatial skyline queries. In: Proceedings of the 32nd international conference on very large data bases, VLDB '06. VLDB endowment, pp 751–762
12. Singh A, Ferhatosmanoglu H, Tosun AS (2003) High dimensional reverse nearest neighbor queries. In: Proc. CIKM
13. Stanoi I, Agrawal D, Abbadi AE (2000) Reverse nearest neighbor queries for dynamic databases. In: Proc. DMKD
14. Tao Y, Papadias D, Lian X (2004) Reverse *k*-NN search in arbitrary dimensionality. In: Proc. VLDB
15. Tao Y, Xiao X, Pei J (2006) Subsky: efficient computation of skylines in subspaces. In: ICDE, pp 65–74
16. Vlachou A, Doulkeridis C, Kotidis Y, Nørvåg K (2010) Reverse top-k queries. In: ICDE, pp 365–376
17. Yang C, Lin K-I (2001) An index structure for efficient reverse nearest neighbor queries. In: Proc. ICDE

Thomas Bernecker is an academic assistant in the database systems and data mining group of Hans-Peter Kriegel at the Ludwig-Maximilians-Universität München, Germany. His research interests include query processing in uncertain databases, spatio-temporal data mining and similarity search in spatial, temporal, and multimedia databases.

Tobias Emrich is an academic assistant in the database systems and data mining group of Hans-Peter Kriegel at the Ludwig-Maximilians-Universität München, Germany. His research interests include query processing in uncertain databases, spatio-temporal data mining and similarity search in spatial, temporal, and multimedia databases.

Hans-Peter Kriegel is a full professor for database systems and data mining in the Department "Institute for Informatics" at the Ludwig-Maximilians-Universität München, Germany and has served as the department chair or vice chair over the last years. His research interests are in spatial and multimedia database systems, particularly in query processing, performance issues, similarity search, high-dimensional indexing as well as in knowledge discovery and data mining. He has published over 300 refereed conference and journal papers and he received the "SIGMOD Best Paper Award" 1997 and the "DASFAA Best Paper Award" 2006 together with members of his research team.

Nikos Mamoulis is a full professor at the Department of Computer Science, University of Hong Kong, which he joined in 2001. In the past, he has worked as a research and development engineer at the Computer Technology Institute, Patras, Greece and as a post-doctoral researcher at the Centrum voor Wiskunde en Informatica (CWI), the Netherlands. During 2008–2009 he was on leave to the Max-Planck Institute for Informatics (MPII), Germany. His research focuses on the management and mining of complex data types, including spatial, spatio-temporal, object-relational, multimedia, text and semi-structured data. He has served on the program committees of over 70 international conferences and workshops on data management and data mining. He was the general chair of SSDBM 2008, the PC chair of SSTD 2009, and he organized the SSTDM 2006 and DBRank 2009 workshops. He has served as PC vice chair of ICDM 2007, ICDM 2008, and CIKM 2009. He was the publicity chair of ICDE 2009. He is an editorial board member for Geoinformatica Journal and was a field editor of the Encyclopedia of Geographic Information Systems.

Matthias Renz is an assistant professor in the database systems and data mining group of Hans-Peter Kriegel at the Ludwig-Maximilians-Universität München, Germany. He finished his PhD thesis on query processing in spatial and temporal data in winter 2006. His research interests include query processing in uncertain databases, spatio-temporal data mining and similarity search in spatial, temporal, and multimedia databases.

Shiming Zhang is a post-doc in the Department of Computer Science, University of Hong Kong. His research interests including efficient processing of skyline queries.

Andreas Züfle is an academic assistant in the database systems and data mining group of Hans-Peter Kriegel at the Ludwig-Maximilians-Universität München, Germany. His research interests include query processing in uncertain databases, spatio-temporal data mining and similarity search in spatial, temporal, and multimedia databases.