Reverse k Nearest Neighbors Query Processing: Experiments and Analysis

Shiyu Yang[†], Muhammad Aamir Cheema[‡], Xuemin Lin[†], Wei Wang[†]
[†]*School of Computer Science and Engineering, The University of New South Wales, Australia*
[‡]*Faculty of Information Technology, Monash University, Australia*
yangs@cse.unsw.edu.au, aamir.cheema@monash.edu, lxue@cse.unsw.edu.au, weiw@cse.unsw.edu.au

ABSTRACT

Given a set of users, a set of facilities and a query facility q, a reverse k nearest neighbors (RkNN) query returns every user u for which the query is one of its k closest facilities. RkNN queries have been extensively studied under a variety of settings and many sophisticated algorithms have been proposed to answer these queries. However, the existing experimental studies suffer from a few limitations. For example, some studies *estimate* the I/O cost by charging a fixed penalty per I/O and we show that this may be misleading. Also, the existing studies either use an extremely small buffer or no buffer at all which puts some algorithms at serious disadvantage. We show that the performance of these algorithms is significantly improved even when a small buffer (containing 100 pages) is used. Finally, in each of the existing studies, the proposed algorithm is mainly compared only with its predecessor assuming that it was the best algorithm at the time which is not necessarily true as shown in our experimental study. Motivated by these limitations, we present a comprehensive experimental study that addresses these limitations and compares some of the most notable algorithms under a wide variety of settings. Furthermore, we also present a carefully developed filtering strategy that significantly improves TPL which is one of the most popular RkNN algorithms. Specifically, the optimized version is up to 20 times faster than the original version and reduces its I/O cost up to two times.

1. INTRODUCTION

A reverse k nearest neighbors (RkNN) query finds every data point for which the query point q is one of its k nearest neighbors [6]. Since q is close to such data points, q is said to have high influence on these points. Consider the example of a shopping market. The residents for which this is one of the k closest markets are its potential customers. In this paper, the objects that provide a facility or service (e.g., shopping market, fuel stations) are called facilities and the objects (e.g., residents, drivers) that use the facility are called users. In this context, RkNN of a query q returns every user u for which q is one of its k closest facilities.

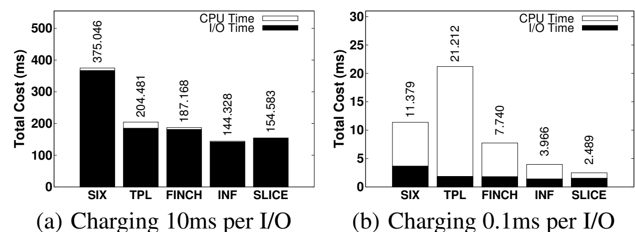
Note that there is a *reverse* relationship between a k nearest neighbor (kNN) query and a RkNN query. While a kNN query helps a user looking for near by facilities, a RkNN query assists the owner of a facility in identifying the users who are potentially interested in her facility. This information might be quite useful for the owner, for instance, these users are likely to be influenced by the advertisements or deals sent to them. The popularity of RkNN queries has also inspired several other reverse spatial queries such as reverse skyline queries [11, 14], reverse top-k queries [8, 24, 32] and reverse furthest neighbors queries [29].

Due to its importance, RkNN query has been extensively studied [2, 4–7, 10, 17–20, 25, 28] ever since it was introduced in [13]. Some of the most notable algorithms are six-regions [18], TPL [20, 21], FINCH [25], InfZone [6, 9] and SLICE [28]. This paper is the first work to present a comprehensive experimental study comparing all these algorithms.

1.1 Motivation

Each of the existing experimental studies reported in the past suffers from at least one of the following serious limitations.

1. The results reported in some of the existing experimental studies (e.g., [20,25]) compare the algorithms on overall cost where the overall cost includes both CPU cost and I/O cost. The I/O cost is *estimated* by charging a fixed penalty per disk I/O, e.g., in [20], the I/O cost is estimated by charging 10ms for each I/O. We argue against using a fixed penalty for each I/O mainly because I/O cost is highly system specific [1] (e.g., type of disk drive used, workload, I/O request re-ordering used by system etc.). For instance, the I/O cost for SSD disks is much lower (e.g., 0.1ms [23]) than the I/O cost for hard disk drives (e.g., tens of milliseconds [16]). For the same reason, the experiments that measure and report the so-called *actual* I/O cost should also be interpreted with caution considering that I/O cost is significantly affected by various factors many of which are system dependent or not easily controllable.



(a) Charging 10ms per I/O (b) Charging 0.1ms per I/O

Figure 1: Charging a fixed penalty per I/O may be misleading

In Figure 1, we compare the performance of the five most notable algorithms: six-regions (displayed as SIX), TPL, FINCH, InfZone (displayed as INF) and SLICE. We issue one thousand RkNN queries ($k = 10$) on a data set that consists of 100,000 facilities and 100,000 users following normal distribution. Figure 1 shows

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 5
Copyright 2015 VLDB Endowment 2150-8097/15/01.

the average total cost (CPU + I/O cost). In Figure 1(a), a penalty of 10ms is applied for each I/O whereas, in Figure 1(b), the penalty is 0.1ms per I/O. Note that the results reported in Figure 1(a) and Figure 1(b) are severely affected by the choice of penalty and are contradictory. For instance, Figure 1(a) shows that the I/O cost is the dominant cost for each algorithm whereas Figure 1(b) demonstrates that the CPU cost is the major cost. Also, Figure 1(a) shows that TPL is better than SIX whereas Figure 1(b) shows that SIX is better than TPL.

2. Some of the algorithms (such as TPL, InfZone and SLICE) access each node of the underlying index (i.e., R*-tree) at most once. On the other hand, six-regions and FINCH require multiple accesses to some nodes of the R*-tree. Hence, the I/O cost of these two algorithms is affected by the size of buffer. We note that RkNN queries are not I/O extensive and the number of unique nodes accessed by the algorithms to answer a RkNN query is small (e.g., less than 100 in our experiments). This implies that a buffer containing only 100 pages can significantly reduce the I/O cost of six-regions and FINCH. Given the large main memory available in modern systems, allocating a buffer containing 100 pages (consuming 400KB memory) should not be a hindrance. However, all of the existing experimental studies either use a very small buffer (10 pages only) or no buffer at all. This adversely affects the performance of six-regions and FINCH especially if a larger penalty is charged per I/O.

3. In each of the existing experimental studies, the proposed algorithm is only compared with its predecessor assuming that the predecessor was the state-of-the-art algorithm at that time. The two limitations mentioned above aggravate this because, as argued above, the predecessor may not necessarily be the best algorithm at the time. For instance, FINCH and InfZone were not compared with SIX assuming that it is outperformed by TPL. However, we observe that SIX is faster than TPL in terms of CPU cost for most of the experimental settings. Hence, there is a need to conduct a comprehensive experimental evaluation that compares all notable algorithms on a range of data sets under a wide variety of settings.

1.2 Major Contributions

Comprehensive experimental study. To the best of our knowledge, we are the first to present a comprehensive experimental study comparing the most notable RkNN algorithms. Instead of charging a fixed penalty for each I/O, we compare the algorithms using two different metrics: i) number of I/Os and ii) CPU time. This choice has several advantages as described below.

Firstly, it avoids the possible distortion of the results caused by charging a fixed penalty per I/O. At the same time, it allows the interested readers to estimate the total cost (by charging a fixed penalty) if they really want to do so. Secondly, it makes it easy for the users to choose algorithms of their choice depending on whether they prefer algorithm with lower I/O cost, lower CPU cost or a trade-off between the two. For instance, modern computers have sufficiently large main memory to store the data sets containing several million data points. The systems that can afford to keep the data sets in main memory may prefer the algorithm with the lowest CPU cost ignoring the I/O cost altogether. On the other hand, a user may prefer the algorithm with lower I/O cost if she does not want the data sets to occupy main memory and has a hard disk drive with high I/O cost.

We also present an imaginary algorithm that assumes the existence of an oracle and achieves the lower bound I/O cost for the case when data sets are indexed by R*-trees. We compare the I/O cost of the existing algorithms with the lower bound I/O cost which helps in identifying the room for further improvement.

Improved version of TPL. TPL is arguably the most popular algorithm to answer RkNN queries and several follow up algorithms are inspired by the basic idea used in TPL. Our experimental study demonstrates that TPL is outperformed by most of the algorithms both in terms of CPU cost and I/O cost. This is mainly because the filtering strategy used by TPL is quite expensive. We propose an improved version of TPL (called TPL++) that replaces the original filtering technique with a carefully developed cheaper yet more powerful filtering strategy and significantly improves its performance. Specifically, TPL++ is up to 20 times better than TPL in terms of CPU cost and up to 2 times better in terms of number of I/Os. Our experimental study demonstrates that TPL++ is one of the best algorithms especially in terms of number of I/Os.

The rest of the paper is organized as follows. In Section 2, we present the problem definition, scope of the paper, algorithmic framework and terminology used throughout the paper. The algorithms that are compared in this paper are described in Section 3. A comprehensive experimental study is reported in Section 4. Section 5 concludes the paper.

2. PRELIMINARIES

2.1 Problem Definition

RkNN queries are classified [28] into *bichromatic RkNN queries* and *monochromatic RkNN queries*.

Bichromatic RkNN Queries. Consider a set of facilities F and a set of users U. Given a query facility q (not necessarily in F), a bichromatic RkNN query returns every user $u \in U$ for which q is one of its k-closest facilities among the facilities in $\{q \cup F\}$.

Monochromatic RkNN Queries. Given a set of facilities F and a query facility q (not necessarily in F), a monochromatic RkNN query returns every facility $f \in F$ for which q is one of its k-closest facilities among the facilities in $\{q \cup F - f\}$.

2.2 Scope

RkNN queries have been extensively studied under different settings such as static RkNN queries [6, 18, 25], continuous RkNN queries [3, 7], probabilistic RkNN queries [4, 5], RkNN queries on graphs [22, 31], metric spaces [22] and adhoc spaces [30] etc. However, in this paper, we focus on answering RkNN queries in Euclidean space. Since most of the applications of RkNN queries are in location-based services, almost all of the existing techniques (e.g., six regions [18], FINCH [25], InfZone [6], and SLICE [28] etc.) focus on two dimensional location data. Therefore, we focus on comparing the performance of the algorithms on two-dimensional data sets. Also, the bichromatic version of RkNN queries have more applications in real world scenarios. Therefore, the main focus of our experimental study is on bichromatic queries. Nevertheless, we also present some results for monochromatic queries and remark that the trends are similar.

Although most of the existing algorithms can be applied on any branch and bound data structure, all of these algorithms assume that the data sets are indexed by R-tree or its variants such as R*-tree. Following this, we also assume that both the facility and user data sets are indexed by two R*-trees. The R*-tree that indexes the set of facilities (resp. users) is called facility (resp. user) R*-tree.

2.3 Framework and Terminology

We say that a facility f *prunes* a point p if $dist(p, f) < dist(p, q)$. Note that a point p that is pruned by at least k facilities cannot be the RkNN of q because q cannot be one of its k closest facilities. We say that a point p is *filtered* if p can be pruned by at least k facilities. Given an entry e (e.g., a node of R*-tree), we say that a

facility f prunes an entry e if f prunes every point p in e. Similarly, we say that an entry e is filtered if e is pruned by at least k facilities.

Each algorithm described in this paper has two phases namely *filtering* and *verification*.

1. Filtering. In the filtering phase, each algorithm uses the set of facilities to filter the search space that cannot contain any RkNN of the query. Since using all of the facilities may be prohibitively expensive, the algorithms choose some of the facilities for filtering the space. These facilities are called filtering facilities and the set containing these facilities is called the filtering set (denoted as S_{fil}).

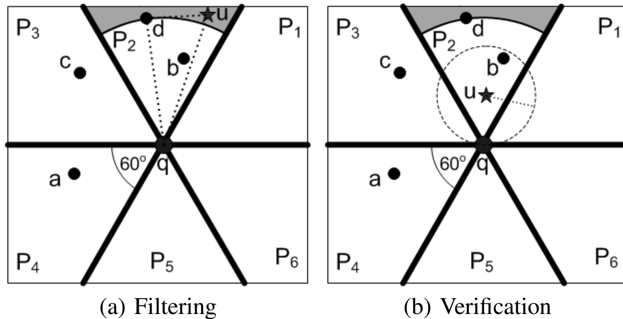
2. Verification. In the verification phase, the users that cannot be filtered using S_{fil} are retrieved. These are the possible RkNNs and are called the candidate users. Each of these candidates is then verified by confirming whether it is a RkNN or not.

3. ALGORITHMS

3.1 Six-regions

3.1.1 *Filtering*

Stanoi *et al.* [18] propose a six-region based approach that partitions the whole space centred at the query q into six equal regions of 60° each (P_1 to P_6 in Figure 2). The k-th nearest facility of q in each region defines the area that can be filtered. In other words, assume that d_i^k is the distance between q and its k-th nearest facility in a region P_i. Then any user u that lies in P_i and lies at a distance greater than d_i^k from q cannot be the RkNN of q.



(a) Filtering (b) Verification

Figure 2: Illustration of six-regions ($k = 2$)

Figure 2 shows a RkNN ($k = 2$) query q and four facilities a to d. In region P_2, d is the second nearest facility of q and the shaded area can be filtered, i.e., only the users that lie in the white area can be the RkNNs. A user u that lies in the shaded area cannot be RkNN because it is guaranteed to be closer to both b and d than q. For instance, this can be proved for the facility d and the user u using the triangle $\triangle qdu$ (see Figure 2(a)). Since $\angle dqu \leq 60^\circ$ and $\angle qdu \geq 60^\circ$, $dist(u,d) \leq dist(u,q)$.

3.1.2 *Verification*

As stated earlier, a user u that lies in a partition P_i cannot be RkNN if $dist(u,q) > d_i^k$. In the verification phase, the candidate users are retrieved by visiting user R*-tree and filtering every entry e for which $mindist(e,q) > d_i^k$. Each candidate user is then verified by issuing a *boolean range query*[1] centered at u with radius $dist(u,q)$. A boolean range query returns true if and only if there are at least k facilities that lie within the circle centered at u with radius $dist(u,q)$. Note that a user u is a RkNN if and only if the boolean range query returns false.

Consider the example of Figure 2(b). The user u is a R2NN of q because the circle centered at u with radius $dist(u,q)$ (the dotted circle) contains only one facility. Since the algorithm issues a boolean range query for each candidate user u, it requires traversing the facility R*-tree once for each candidate user.

3.2 TPL

3.2.1 *Filtering*

Tao *et al.* [20,21] propose TPL which is arguably the most popular algorithm for RkNN queries. They were the first to use the concept of *half-space* pruning for RkNN queries and inspired many follow up works (e.g., [4–6,25]). Given a facility f and a query q, a perpendicular bisector $B_{f:q}$ between f and q divides the space into two halves. Let $H_{f:q}$ denote the half-space that contains f and $H_{q:f}$ denote the half-space that contains q. Every point p that lies in $H_{f:q}$ satisfies $dist(p,f) < dist(p,q)$. In other words, f prunes every point p that lies in $H_{f:q}$.

Consider the example of Figure 3(a) where a query q and four facilities a to d are shown. The point p can be pruned by the facility a because p lies in $H_{a:q}$ and, therefore, $dist(p,a) < dist(p,q)$. Note that a point p that is pruned by at least k half-spaces cannot be the RkNN and can be filtered. Assuming $k = 2$, the point p can be filtered because it is pruned by both $H_{a:q}$ and $H_{c:q}$.

Starting from the root node, the filtering algorithm of TPL iteratively accesses the entries of the facility R*-tree from a heap in ascending order of their minimum distances from q. The accessed facilities are used for filtering the search space. If an accessed entry e can be filtered (i.e., e is pruned by at least k facilities), it is ignored. Otherwise, if e is an intermediate or leaf node, its children are inserted in the heap. On the other hand, if e is a facility and cannot be filtered, it is inserted in the filtering set S_{fil} and its half-space is used to filter the search space. The filtering algorithm terminates when the heap becomes empty.

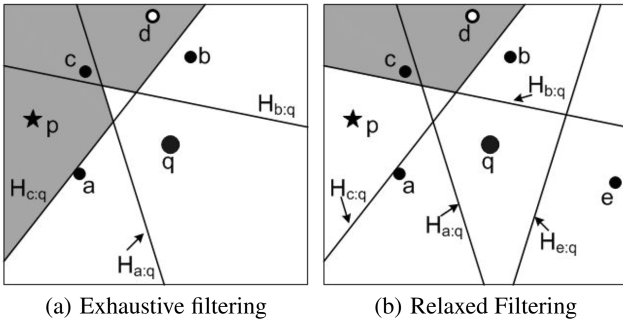
In Figure 3(a), assume that the filtering algorithm iteratively accesses the facilities in the order b, c, a and d. When the facilities b, c and a are accessed (i.e., $S_{fil} = \{b,c,a\}$), the filtered area is defined by the half-spaces $H_{b:q}$, $H_{c:q}$ and $H_{a:q}$. If $k = 2$, the shaded area can be filtered because every point in it lies in at least two half-spaces. Note that when d is accessed, it can be filtered using the filtering set. Hence, d is not inserted in S_{fil}.

An important operation in TPL is to determine whether an entry (a node of R*-tree or a data point) can be filtered using a set of filtering facilities S_{fil} or not. An exhaustive filtering strategy is the following. Let S_{fil} be the filtering set containing $m \geq k$ facilities and $\{\rho_1, \cdots \rho_k\}$ be any subset of S_{fil} containing k facilities. The subset filters the space $\cap_{i=1}^{k} H_{\rho_i:q}$ because each point p in this space is pruned by each of the k facilities in this subset. The exhaustive filtering strategy is to consider each such subset and check if the entry can be filtered using these subsets or not. However, the total number of such subsets is $\binom{m}{k}$ and considering all these subsets may be prohibitively expensive . Hence, TPL compromises on the filtering power and uses the following less expensive filtering strategy (called relaxed filtering strategy hereafter).

First, TPL sorts the facilities in S_{fil} on their Hilbert values. Let the sorted S_{fil} be $\{f_1, \cdots, f_m\}$. The relaxed filtering considers m subsets $\{f_1, \cdots f_k\}, \{f_2, \cdots, f_{k+1}\}, \cdots, \{f_m, \cdots, f_{k-1}\}$. Note that the total filtering cost is $O(km)$ because m subsets each containing k facilities are to be considered.

Consider the example of Figure 3(a) and assume that $S_{fil} = \{a,b,c\}$. The relaxed filtering considers the subsets $\{a,b\}$, $\{b,c\}$

[1]The original algorithm proposed to use kNN queries for verification. However, since it is well known that boolean range queries are more efficient [9,25], we use these to verify the candidates.



(a) Exhaustive filtering (b) Relaxed Filtering

Figure 3: Illustration of TPL filtering ($k = 2$)

and $\{c, a\}$ and filters the same area as the exhaustive filtering (i.e., the shaded area). This is because the number of maximum possible subsets is equal to m in this case, i.e., $\binom{m}{k} = \binom{3}{2} = 3 = m$. Next, we show that adding a new facility in S_{fil} may reduce the filtered area and the points that could be filtered before adding this facility may not be filtered anymore Assume that the facility e is inserted and the sorted S_{fil} is $\{a, b, c, e\}$. The relaxed filtering algorithm uses $\{a, b\}$, $\{b, c\}$, $\{c, e\}$ and $\{e, a\}$ to filter the search space. Figure 3(b) shows the shaded area that can be filtered using these subsets. Note that the filtered area has become smaller and the point p cannot be filtered anymore.

To filter an entry e (a node or data point), TPL uses an algorithm called `kTrim`. For a subset of S_{fil} containing k consecutive facilities, the algorithm trims the part of the entry that can be filtered by this subset. `kTrim` uses each of the m subsets and iteratively trims the entry. At any stage, if the whole entry is trimmed, the node or data point can be filtered and the algorithm stops

3.2.2 Verification

TPL iteratively accesses the entries of the user R*-tree and filters them using S_{fil}. The users that cannot be filtered are inserted in the candidate set. Unlike six-regions, TPL does not issue boolean range queries to verify the candidates. Instead, TPL uses a smart strategy that requires each node of the R*-tree to be visited at most once. Specifically, the nodes and points of the facility R*-tree that are encountered during the filtering phase are kept in a set P. The verification algorithm runs in *rounds*. In each round, one of the nodes in P is opened and its children are inserted in P. The node is selected based on how many candidates it can potentially verify. During each round, the nodes and points in P are used to identify the candidates that can be verified using P, i.e., can be confirmed as RkNN or guaranteed not to be RkNN. Such candidates are verified and removed from the candidate set. The algorithm stops when the candidate set becomes empty.

3.3 TPL++

In this section, we present an optimized version of TPL (called TPL++) that is based on two optimizations. We remark that the proposed optimizations work for arbitrary dimensionality and are expected to improve TPL even for higher dimensionality.

Optimization 1. The first optimization is an improved filtering strategy that takes $O(m)$, in contrast to $O(km)$ required by `kTrim`, and filters more entries than `kTrim`. Let $\{f_1, \cdots, f_j\}$ be a subset of the filtering set S_{fil} containing $j \geq 1$ facilities. Note that every point p that lies in $\cup_{i=1}^{j} H_{f_i:q}$ is pruned by *at least* one facility in this subset. Consider the example of Figure 4 where the subset $\{a, b\}$ prunes $H_{a:q} \cup H_{b:q}$ (the shaded area in Figure 4). The entry e (the rectangle) can be pruned because each point in this rectangle is either pruned by the facility a or by the facility b.

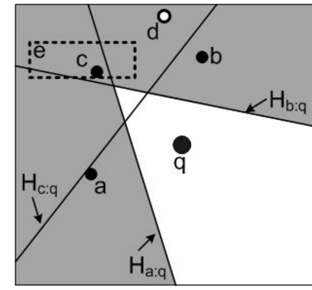


Figure 4: Optimized Filtering

The basic idea of the algorithm is that an entry can be filtered if it is pruned by at least k such subsets where these subsets may be of different sizes but all these subsets must be disjoint (i.e., each facility appears in at most one subset). In the example of Figure 4, the entry e can be filtered because it is pruned by two subsets $\{a, b\}$ and $\{c\}$.

Algorithm 1: isFiltered(S_{fil}, e)

Input : S_{fil}: the filtering set, e: the entry to be filtered
Output : Return true if the entry can be filtered, otherwise return false
1 counter $\leftarrow 0$;
2 $e^{tmp} \leftarrow e$;
3 **for each** facility $f \in S_{fil}$ **do**
4   **if** e lies completely in $H_{f:q}$ **then**
5   counter ++;
6   **else**
7   e^{tmp} = Trim($e^{tmp}, H_{f:q}$) `// algorithm in [12];`
8   **if** $e^{tmp} = \emptyset$ **then** `// if the whole` e^{tmp} `is pruned`
9   counter ++;
10       $e^{tmp} \leftarrow e$
11   **if** counter $= k$ **then**
12   return true;
13 return false;

Algorithm 1 presents the details of our optimized filtering strategy. The basic idea is to use a counter that records the number of subsets (seen so far) that prune the entry. The facilities in S_{fil} are iteratively accessed and the entry e is tried to prune using the facilities seen so far. Whenever e is pruned, the counter is incremented. The entry can be filtered if the counter is at least k.

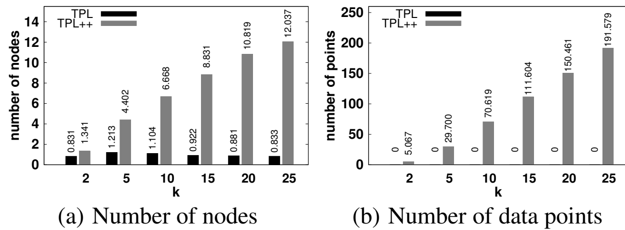
Note that some facilities may completely prune the entry e and some may prune only a part of the entry. e^{tmp} is used to store the part of the entry e that cannot be pruned by the current subset of facilities and is initialized as e (line 2 and line 10). An accessed facility f is used to trim the part of e^{tmp} that can be pruned by the facility f. Specifically, similar to TPL, the clipping algorithm [12] is used which uses the half-space $H_{f:q}$ and returns the part of e^{tmp} that cannot be pruned by f (line 7). The facilities are accessed iteratively and e^{tmp} is iteratively trimmed. If e^{tmp} becomes empty, it implies that the entry e is pruned by a subset of the facilities. Hence, the counter is incremented and e^{tmp} is initialized to e (lines 8 to 10).

Note that a facility f that completely prunes an entry e is handled differently (line 4). Specifically, instead of trimming e^{tmp} using such a facility, the counter is incremented by one. This is because $\{f\}$ itself is a subset that can prune e and it is suboptimal to include f in a subset that can prune e. Consider the example of Figure 4 and assume that the objects in S_{fil} are ordered as $\{a, c, b\}$. Algorithm 1 can filter e because it is pruned by $\{a, b\}$ and $\{c\}$. If the facility c is not handled differently (i.e., lines 4 to 6 are removed), then e cannot be filtered because only the subset $\{a, c\}$ prunes it.

Remark. Filtering power of Algorithm 1 is lower than the exhaustive algorithm and it may not be able to filter every R*-tree node that can be filtered by the exhaustive filtering. However, it can be guaranteed that every data point (i.e., a facility or a user) that can be filtered by exhaustive filtering can also be filtered by Algorithm 1.

LEMMA 1 : Every data point p that can be filtered by the exhaustive filtering can also be filtered by Algorithm 1.

PROOF. If a point p is filtered by the exhaustive algorithm, then there must exist a subset $\{\rho_1, \cdots, \rho_k\}$ such that p lies in $\cap_{i=1}^{k} H_{\rho_i:q}$. In other words, each of $H_{\rho_i:q}$ contains the point p. Note that Algorithm 1 will increase the counter for each of such facility ρ_i (at line 4). Since there are at least k such facilities, Algorithm 1 can also filter p. □



(a) Number of nodes (b) Number of data points

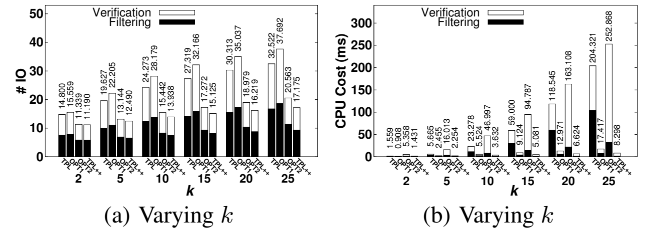
Figure 5: Comparison of filtering power (TPL vs TPL++)

In Figure 5, we compare the relative filtering power of relaxed filtering used by TPL and Algorithm 1 used by TPL++. We did not consider the exhaustive filtering because the cost is prohibitively expensive, e.g., for $k = 10$ and 30 facilities in S_{fil}, the total number of possible subsets is more than 30 million. We use TPL to answer $1,000$ RkNN queries on the data sets that contain $100,000$ facilities and $100,000$ users (following normal distribution) and report average result per query. Whenever TPL checks whether an entry can be filtered or not (using relaxed filtering), we also test whether Algorithm 1 could filter the entry or not. This ensures that the same filtering set S_{fil} is used by both of the strategies.

Figure 5 shows the average number of entries (nodes in Figure 5(a) and points in Figure 5(b)) pruned by one filtering strategy but not by the other, e.g., the bar corresponding to TPL demonstrates the number of nodes that are filtered by relaxed filtering but not by Algorithm 1. Note that although Algorithm 1 is less expensive, it has more filtering power as shown in Figure 5(a) and Figure 5(b). Also, the difference in the filtering power increases as k increases. The results for $k = 1$ are not shown because both filtering strategies become the same for $k = 1$.

Optimization 2. Note that the size of S_{fil} directly affects the cost of filtering as well as the filtering power of the algorithm. As stated earlier, TPL only inserts a facility in S_{fil} if it cannot be filtered using the existing set S_{fil}, e.g., in Figure 3(b), the facility d is not inserted in S_{fil}.

Since we propose a cheaper filtering strategy, we propose to include more facilities in S_{fil}. Specifically, similar to TPL, if an accessed entry e is an intermediate or leaf node and can be filtered, we ignore it. However, if an accessed entry e is a facility, we insert it in S_{fil} regardless of whether it can be filtered or not. Note that this increases the size of S_{fil} for TPL++ and, in effect, increases the filtering cost. However, it results in a much better filtering power which results in requiring fewer calls to the filtering algorithm. Also, in contrast to TPL, we do not need to check whether the facility can be filtered or not which further reduces the number of times filtering algorithm is called. As a consequence, the I/O cost as well as the CPU cost of the algorithm is reduced.



(a) Varying k (b) Varying k

Figure 6: Effect of optimizations: 100K facilities and 100K users following Normal distribution

Figure 6 shows the effect of the two proposed optimizations. Specifically, OPT1 is the algorithm that only uses optimization 1 (optimized filtering), OPT2 is the version where only the second optimization is applied. TPL++ is the algorithm when both of the optimizations are applied. Figure 6(a) shows the I/O cost of each algorithm. Note that the I/O cost of OPT2 is smaller than the I/O cost of TPL although both algorithms use the same filtering strategy. This is because OPT2 has a larger filtering set S_{fil} and hence can filter more entries. The I/O cost of OPT1 is larger than TPL. This is because the optimized filtering used by OPT1 filters more facilities which results in a smaller filtering set S_{fil}. Since S_{fil} is smaller for OPT1, the total number of entries pruned by OPT1 are smaller. Note that TPL++ has the lowest I/O cost.

Figure 6(b) shows the CPU cost of each algorithm. OPT1 is more efficient than TPL because it employs a cheaper filtering strategy. Although OPT2 uses the same filtering strategy as TPL, it is significantly more expensive. This is because S_{fil} is much larger for OPT2 and the filtering cost of OPT2 and TPL increases rapidly with increase in S_{fil}. In general, TPL++ is the most efficient algorithm mainly because it employs a cheaper filtering strategy and uses a larger S_{fil} which results in more entries being filtered. For $k = 2$, OPT1 is better than TPL++. This is because a larger S_{fil} used by TPL++ implies a more effective but, at the same time, a more expensive filtering. This results in an overall relatively poor performance (as compared to OPT1) for smaller value of k.

3.4 FINCH

3.4.1 Filtering

Wu *et al.* [25] propose an algorithm called FINCH that aims at improving upon TPL by using a cheaper filtering strategy. Specifically, instead of using the subsets to filter the entries, they use a convex polygon that encloses the unfiltered area. Any entry that lies outside the polygon can be filtered.

Consider the example of Figure 3(a) where the white area is the unfiltered area. FINCH approximates this area by a convex polygon (see the white area in Figure 7 with boundary shown in broken lines). Any entry that lies outside this polygon can be filtered, i.e., the shaded area of Figure 7 can be filtered.

The cost of computing the convex polygon is high. Specifically, whenever a new facility is added to S_{fil}, it takes $O(m^2)$ to compute the convex polygon where m is the number of facilities in the filtering set S_{fil}. However, filtering of FINCH is more efficient than TPL because containment of a point can be done in logarithmic time for convex polygons. Hence, a point can be filtered in $O(\log\ m)$. Filtering a node takes $O(m)$ because it may require computing the intersection of the rectangle with the convex polygon.

3.4.2 Verification

The users that lie inside the convex polygon are identified and are inserted in the candidate set. Similar to the six-regions approach, each candidate user is verified using the boolean range query.

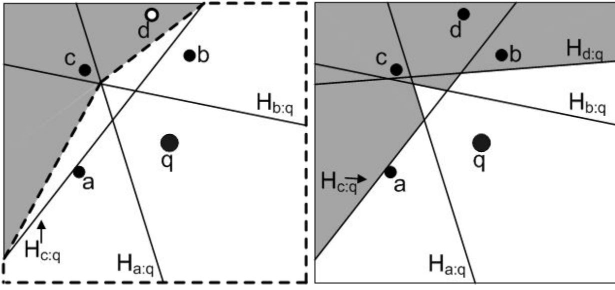


Figure 7: FINCH ($k = 2$) **Figure 8: InfZone ($k = 2$)**

3.5 InfZone

3.5.1 *Filtering*

Cheema *et al.* [6, 9] propose InfZone which uses the concept of *influence zone* to significantly improve the verification phase. Influence zone is the area such that a point p is a RkNN of q if and only if p lies inside this area. Once influence zone is computed, RkNN query can be answered by locating the users lying inside it.

A naïve approach to construct the influence zone is to draw the half-spaces of *all* the facilities. The area that is filtered by less than k facilities corresponds to the influence zone. For instance, in Figure 8, the half-spaces between q and all facilities are drawn ($H_{a:q}$, $H_{b:q}$, $H_{c:q}$, and $H_{d:q}$). The shaded area can be filtered and the white area is the influence zone. Recall that FINCH and TPL did not consider $H_{d:q}$ because, when d is accessed, it is found to lie in the filtered area and is ignored.

The authors present several properties to reduce the number of facilities that must be considered in order to correctly compute the influence zone. The algorithm initializes the influence zone to be the whole data space (which is a rectangle). Whenever a facility f is accessed, its half-space $H_{f:q}$ is used to update the influence zone, e.g., by removing the part that is pruned by at least k facilities. The key observation is that an entry e (a node or a facility) is not required to correctly compute the influence zone if $mindist(e, v) > dist(v, q)$ for every vertex v of the current influence zone. Hence, the algorithm only considers the entries that do not satisfy this condition. Note that checking whether the facility entry is required or not takes $O(m)$ because it was shown that the number of vertices of the influence zone is $O(m)$.

Updating the influence zone when a new facility is inserted in S_{fil} takes $O(m^2)$ where m is the total number of facilities in S_{fil}. In an extended version [9], this cost was improved to $O(km)$.

3.5.2 *Verification*

By definition of the influence zone, a point p is a RkNN if and only if p is inside the influence zone. Hence, the entries of user R*-tree are accessed iteratively and the entries that do not overlap with the influence zone are ignored. The users that lie inside the influence zone are reported as RkNN.

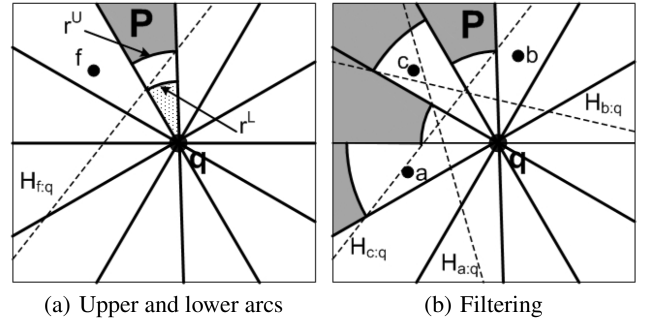
It was shown that the influence zone is a star-shaped polygon [15] and the point containment can be done in logarithmic time to the number of edges of the star-shaped polygons, i.e., the cost to verify a user is $O(\log\ m)$. To filter a user node (i.e., a rectangle), the intersection between the rectangle and influence zone is required which takes $O(m)$. To speed up checking whether an entry (a node or point) overlaps the influence zone or not, InfZone is approximated by two circles: one completely contains the influence zone and the other is completely contained by it. The overlap of each entry is first checked against these circles to see if the overlap with the influence zone is required or not.

3.6 SLICE

3.6.1 *Filtering*

Inspired by the cheap (although less effective) filtering strategy used by six-regions, Yang *et al.* [28] proposed an algorithm called SLICE which improves the filtering power of six-regions approach while utilizing its strength of being a cheaper filtering strategy.

SLICE divides the space around q into multiple equally sized regions. Their experimental study demonstrated that the best performance is achieved when the space is divided into 12 equally sized regions. Figure 9(a) shows an example where the space is divided into 12 equally sized regions. Consider the half-space $H_{f:q}$ and the partition P. The shaded area in partition P can be pruned by $H_{f:q}$ and the dotted area in the partition P cannot be pruned by $H_{f:q}$. The shaded area is defined by an arc centered at q with radius r^U (as shown in Figure 9(a)). This arc is called *upper arc* of f w.r.t. P and is denoted as $r^U_{f:P}$ (or r^U if clear by context). The arc that defines the dotted area is called the *lower arc* of f w.r.t. P and is denoted as $r^L_{f:P}$ (or r^L if clear by context).



(a) Upper and lower arcs (b) Filtering

Figure 9: Illustration of filtering by SLICE ($k = 2$)

It is easy to see that a point p in the partition P can be pruned by the facility f if p lies *outside* its upper arc, i.e., $dist(p, q) > r^U_{f:P}$. For instance, a point p that lies in the shaded area of Figure 9(a) can be pruned. Also, note that a point $p \in P$ cannot be pruned by f if p lies *inside* its lower arc, i.e., $dist(p, q) < r^L_{f:P}$. For instance, f cannot prune a point that lies in the dotted area in Figure 9(a).

During the filtering phase, SLICE maintains k-smallest upper arcs for each partition P_i. The k-th smallest upper arc in a partition P is called its bounding arc and its radius is denoted as r^B_P. Note that any point p that lies in P and $dist(p, q) > r^B_P$ can be pruned by at least k facilities and can be filtered. Figure 9(b) shows the filtering phase of SLICE using three facilities a, b and c (assuming $k = 2$). Figure 9(b) also shows bounding arcs for some of the partitions and the shaded area can be filtered. Note that, for a point p in a partition P, checking whether p can be filtered or not takes $O(1)$. To filter a node entry e that overlaps with multiple partitions, $mindist(q, e)$ is compared with the bounding arc of each overlapping partition. Hence, the cost is $O(t)$ in the worst case where t is the total number of partitions.

3.6.2 *Verification*

In the verification phase, every user u that lies within the bounding arc of its partition is retrieved and inserted in the candidate set. Next, we describe how to verify the candidate users.

As stated earlier, a facility f cannot prune the user u if u lies inside the lower arc of f w.r.t. P, i.e., $dist(u, q) < r^L_{f:P}$. In other words, a facility f can prune a candidate user in P only if its lower arc is smaller than the bounding arc of this partition. Such a facility is called the *significant facility* for the partition P. In the

Filtering Phase	Six-regions	TPL	TPL++	FINCH	InfZone	SLICE
Filter a facility node	$O(1)$	$O(km)$	$O(m)$	$O(m)$	$O(m)$	$O(t)$
Filter a facility point	$O(1)$	$O(km)$	$O(m)$	$O(\log m)$	$O(m)$	$O(t)$
Adding a new facility in S_{fil}	$O(\log k)$	$O(\log m)$	$O(\log m)$	$O(m^2)$	$O(m^2)$	$O(t \log m)$
Verification Phase						
Filter a user node	$O(1)$	$O(km)$	$O(m)$	$O(m)$	$O(m)$	$O(t)$
Filter a user point	$O(1)$	$O(km)$	$O(m)$	$O(\log m)$	$O(\log m)$	$O(1)$
Verify a candidate	boolean range query	concurrent range query	concurrent range query	boolean range query	$O(\log m)$	$O(m)$ expected cost $O(k)$
Expected number of candidates	$\frac{6k\|U\|}{\|F\|}$	$\frac{k\|U\|}{\|F\|}$ to $\frac{6k\|U\|}{\|F\|}$	$\frac{k\|U\|}{\|F\|}$ to $\frac{3.1k\|U\|}{\|F\|}$	$\frac{k\|U\|}{\|F\|}$ to $\frac{6k\|U\|}{\|F\|}$	$\frac{k\|U\|}{\|F\|}$	$< \frac{3.1k\|U\|}{\|F\|}$

Table 1: Comparison of computational complexities. m: number of facilities in S_{fil}, t: number of partitions for SLICE, $|F|$: total number of facilities in the data set, $|U|$: total number of users in the data set

filtering phase, for each partition P, SLICE creates a *significant list* that contains all significant facilities of the partition P. To facilitate efficient verification, the facilities in the significant list are sorted in ascending order of the radii of their lower arcs.

To check whether a user u is a RkNN or not, the algorithm iteratively accesses the facilities in the significant list. A counter is maintained that counts the number of facilities that prune the user u. If the accessed facility f prunes the user u, the counter is incremented. If the counter becomes equal to k, the user u is filtered. At any stage, if the radius of the lower arc of the accessed facility f is larger than $dist(u, q)$, the algorithm terminates. This is because all unseen facilities in the significant list also have lower arcs larger than $dist(u, q)$ and hence cannot prune u.

It was shown that the expected size of significant list is $O(m)$. Hence, verifying a candidate user takes $O(m)$. It was also shown that the expected cost of verifying a user is $O(k)$, i.e., the algorithm is expected to verify a user by checking first k facilities in the significant list.

3.7 Summary of computational costs

Table 1 summarizes the complexities of different operations for each of the algorithms. Note that different algorithms use different S_{fil}, therefore, the value of m is different for each algorithm. We observe that the value of m does not generally depend on the data set size and increases linearly with k. We remark that although Table 1 presents a reasonable picture of the cost of different phases of the algorithms, it is not conclusive and should be seen only as a guide. This is because the cost of each algorithm depends not only on the size of its S_{fil} but also on the number of times each operation is called. Next, we summarize the complexities that have not been discussed in detail earlier.

Whenever a new facility is added in S_{fil}, FINCH and InfZone need to update the convex polygon and the influence zone, respectively. This takes $O(m^2)$ [6, 25]. It was shown that it is possible to update the influence zone in $O(km)$ [9]. However, we remark that this result is more of theoretical interest and does not necessarily improve the performance because m is in practice equal to Ck where C is a small constant. Six-regions need to maintain the k-th nearest facility in each region whenever a new facility is encountered and this takes $O(\log k)$. TPL and TPL++ add the facility in S_{fil} in the sorted order of its Hilbert value and it takes $O(\log m)$. SLICE needs to maintain the set of significant facilities in sorted order for each of the t partitions and this requires $O(t \log m)$ for each new facility f.

The verification cost of InfZone and SLICE is the smallest whereas six-regions and FINCH have the most expensive verification. TPL and TPL++ aim to reduce the I/O cost of the verification phase and use a smart strategy to verify a set of candidates. The cost is somewhat similar to issuing a set of concurrent boolean range queries.

The table also summarizes the expected number of candidates for each algorithm assuming that the users and facilities data sets follow uniform distribution. It was shown in [6] that the expected number of candidates for InfZone is $k|U|/|F|$ where $|U|$ and $|F|$ are the total number of users and facilities $|F|$, respectively. It is easy to show that the expected number of candidates for six-regions is $6k|U|/|F|$, e.g., in each of the six regions, a k-th nearest facility defines the filtered space and, assuming $|U| = |F|$, the number of users inside the unfiltered space is also k for each region. Hence, the expected number of candidates is $6k$ when $|U| = |F|$.

It was shown in [20] that TPL filters more space than six-regions. Hence, the expected number of candidates is smaller than $6k|U|/|F|$. Furthermore, InfZone has the minimal number of candidates because every candidate is guaranteed to be a RkNN (by definition of the influence zone). Hence, the expected number of candidates for TPL is larger than that of InfZone. The expected number of candidates for SLICE is less than $3.1k|U|/|F|$ as shown in [28]. Finally, since SLICE filters the space by approximating the space pruned by half-spaces, its filtering power is expected to be lower than TPL++. Hence, TPL++ is expected to have less than $3.1k|U|/|F|$ candidates. In our experiments, we find that the number of candidates for TPL++ is very close to that of InfZone.

4. EXPERIMENTS

4.1 Experimental Setup

Algorithms and implementation. We implemented all of the algorithms ourselves and the algorithms use common subroutines to do similar tasks. All algorithms are implemented in C++ and compiled to 32-bit binaries by GCC 4.7.2 with -O2 flag. All the experiments are run on Intel Xeon 2.66 GHz quad core CPU, 4GB main memory running Debian Linux and using ST3450857SS hard disk (450GB 15K RPM SAS 6Gbps).

For each algorithm, we also implemented some simple yet effective optimizations that were not mentioned in the original paper. Furthermore, after finishing our implementation, we obtained the codes from the authors to make sure that our implementation does not have any glitches that may negatively affect the algorithm. The detailed pseudocodes, the source code, data sets, and the scripts to run experiments and draw figures are available online[2]. Below are two notable examples for the optimizations we implemented.

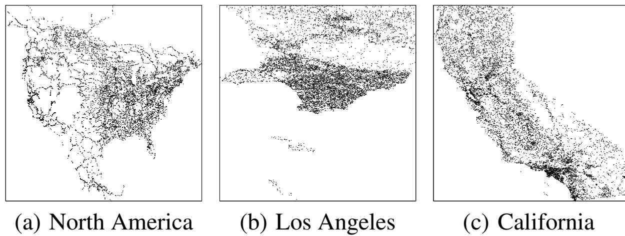
FINCH needs to frequently check whether an entry (a node or point) overlaps with the convex polygon. While computing convex polygon, we compute a minimum bounding rectangle of the

[2]www.aamircheema.com/reproducibility/rknn

polygon as well as a *minimum bounding circle* (centered at q) of the polygon. An entry is guaranteed not to overlap the polygon if it does not overlap its minimum bounding rectangle or minimum bounding circle.

Recall that `kTrim` algorithm used in TPL *trims* the input entry by removing the part of the entry that can be filtered (the returned entry is called output entry). We found that the implementation by the authors repeatedly calls `kTrim` by passing the output entry as the input until either the whole entry is trimmed or the output entry is the same as the input entry. The repeated calls of `kTrim` may eventually filter the entry because at each call of `kTrim` the size of entry is reduced. We also implemented this optimization for TPL because this improves the I/O cost as well as the CPU cost of the algorithm for all experiments.

Data sets. We use three real data sets that consist of points of interest in North America, Los Angeles and California, respectively. The North America [27] data set (denoted as NA) consists of $175,812$ points. The Los Angeles and California data sets (denoted as LA and CA, respectively) are obtained from U.S. census database [26] and are converted from line data sets into point data sets by evenly generating a set of points on each line. LA data set contains 2.6 million points and CA data set contains around 25.8 million points. The existing experimental studies also used these data sets in their experiments: the NA data set was used in the papers proposing TPL, InfZone and SLICE; the LA data set was used by FINCH and TPL; and the CA data set was used by FINCH. Figure 10 shows the distribution of the data points in the three real data sets (using $10,000$ randomly selected points for each data set).



(a) North America (b) Los Angeles (c) California

Figure 10: Real data sets

We also generate synthetic data sets following different data distributions. Due to the similarity in the trends, in this paper, we only report the results for data sets following normal distribution. We conduct experiments on a variety of data sizes ranging from *small* data sets to *large* data sets. Specifically, the small synthetic data set contains 2000 points whereas the medium and the large data sets contain $200,000$ and $20,000,000$ points, respectively.

Each of the real and synthetic data sets is randomly divided into two disjoint sets of points of almost equal sizes. One of the sets corresponds to the set of facilities and the other to the set of users. Each data set is indexed by an R*-tree with the page size of $4,096$ bytes. Average number of children in each node is around 68. We also conducted experiments using different page sizes but observed similar trends.

Similar to existing work, we vary the value of k from 1 to 25 and the default value of k is 10. We also present results for larger values of k (e.g., up to 200). For each data set, one thousand facilities are randomly selected from the facilities data set and these facilities are chosen as the query points. *The results reported in the figures correspond to the average cost per query.* Each query is treated as a fresh run, e.g., the nodes accessed by previous queries are assumed not to be present in the buffer and the computation from the previous queries is not re-used. This choice is made mainly because none of the existing techniques focuses on re-using the information from previous queries and our focus is to evaluate the cost of a *single* RkNN query.

4.2 Effect of buffers

As stated earlier, six-regions and FINCH issue boolean range queries to verify the candidate users. This requires traversing the facility R*-tree once for each boolean range query which may result in a large number of I/Os. In this section, we study the effect of number of buffers on the performance of the algorithms where each buffer corresponds to one page ($4,096$ bytes). Specifically, we change the number of buffers from 1 to 1000. We conduct experiments on CA data set (the largest data set) for $k = 25$ and report the results in Figure 11.

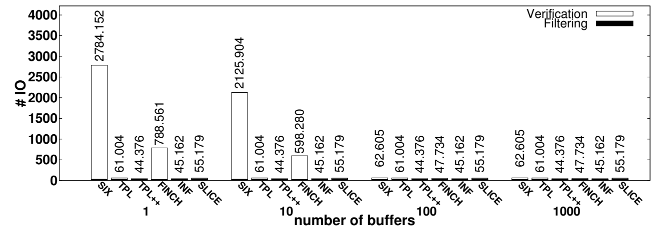


Figure 11: Effect of buffer size (CA data set)

Note that the cost of SIX and FINCH remains unchanged when the number of buffers is 100 or more. This is due to the fact that RkNN queries are not I/O extensive in the sense that each RkNN query requires accessing less than 100 unique nodes of the R*-tree. Note that the other algorithms (TPL, TPL++, InfZone and SLICE) access each node of the R*-tree at most once. Hence, their performance is not affected by the change in buffer size.

As stated earlier, the existing experimental studies either used a buffer containing 10 pages or no buffer at all. Given the increased main memory in modern systems, we believe that it is unfair to these algorithms to use such a small buffer. For the rest of the experiments, we choose 100 buffers (400KB) for each algorithm.

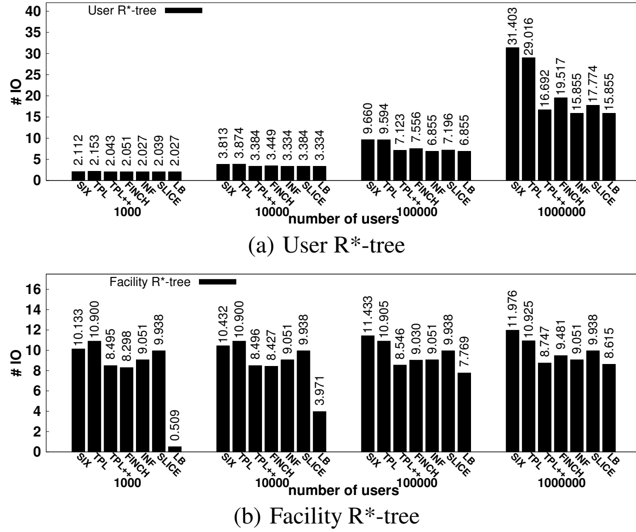
4.3 Comparison with lower bound I/O cost

First, we compare the I/O cost of each algorithm with an imaginary algorithm (called LB algorithm) that assumes the existence of an oracle and achieves lower bound I/O cost, e.g., every algorithm that uses the data sets indexed by R*-trees must access every node that is accessed by the LB algorithm. First, we briefly describe the minimal number of nodes accessed on user R*-tree and facility R*-tree.

Minimal number of accessed nodes on user R*-tree. By the definition of the influence zone, a user u is a RkNN of q if and only if u lies in the influence zone. This implies that every algorithm that correctly computes the results must access every node of the user R*-tree that overlaps with the influence zone. This is because if such a node e is not accessed, the algorithm may miss a user $u \in e$ that lies in the influence zone and is a RkNN.

Minimal number of accessed nodes on facility R*-tree. Let u be a RkNN of the query. Let $|e|$ denote the total number of facilities stored in the sub-tree rooted at e. Every node e of the facility R*-tree for which $mindist(u, e) < dist(u, q)$ and $|e| > k$ must be accessed in order to compute the correct results. This is because if e is not accessed, it is possible that there are at least k facilities in e (or its children) for which $dist(u, f) < dist(u, q)$. In other words, u cannot be guaranteed as a RkNN if e is not accessed.

The LB algorithm. The LB algorithm works as follows. We assume an oracle that computes the influence zone and RkNNs without incurring any I/O cost. In filtering phase, LB algorithm accesses every node e of the facility R*-tree for which $mindist(u, e) < dist(u, q)$ for any RkNN user u. In the verification phase, LB algorithm accesses every node e of the user R*-tree that overlaps with the influence zone.



(a) User R*-tree

(b) Facility R*-tree

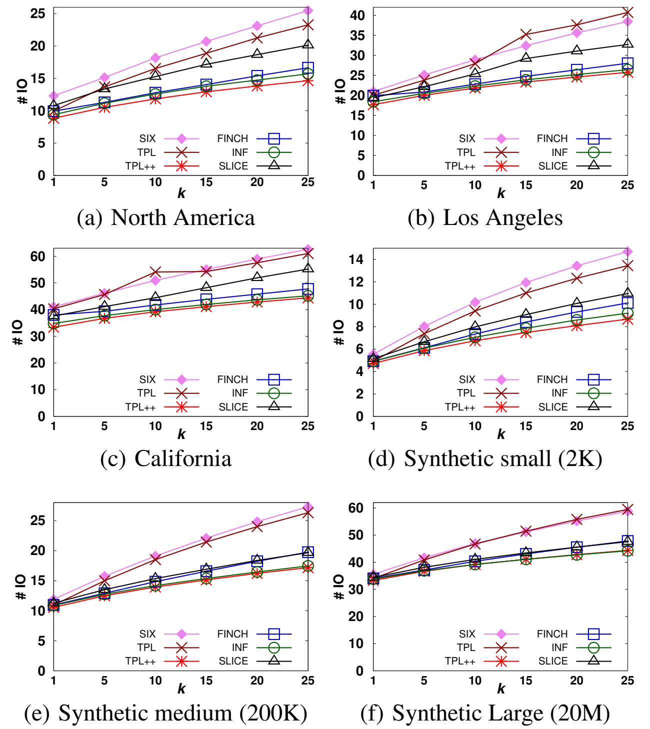
Figure 12: Comparison with lower-bound I/O: Effect of varying $|U|$ ($|F| = 100,000$)

Due to the space limitation, we report only the most interesting result. Specifically, in Figure 12, we use synthetic data sets and fix the number of facilities to 100,000 and vary the number of users.

I/O cost on user R*-tree. Figure 12(a) shows that the number of user R*-tree nodes accessed by InfZone meets the lower bound because, like LB, InfZone also accesses a node if and only if it overlaps with the influence zone. The I/O cost of TPL and SIX are the highest due to poor filtering power. The gap between InfZone/LB and the other algorithms increases as the number of users increases. The reason is as follows. Since the number of facilities does not change, the area filtered by each algorithm remains unchanged. However, as the size of users data set increases, the number of small user R*-tree nodes (in terms of area) increases significantly which results in a larger difference between the number of nodes filtered by the influence zone and the other algorithms.

I/O cost on facility R*-tree. Figure 12(b) shows the number of facility R*-tree nodes accessed by each algorithm. None of the algorithms meet LB cost. Among the competitors, TPL++ and FINCH access minimum number of nodes. SIX and TPL access more nodes because of their low filtering power. InfZone and SLICE access more nodes because these algorithms aim at accessing all the facilities that *may* be required in the verification phase. For the same reason, the number of facility R*-tree nodes accessed by InfZone and SLICE remain the same regardless of the size of users data set. On the other hand, the I/O cost of other algorithms changes as the number of users increases. This is because these algorithms access facility R*-tree to verify the candidates and the number of candidates increases with the number of users.

Note that the gap between LB and the other algorithms is larger for smaller number of users. This is mainly because filtering phase of each algorithm is totally independent of the verification phase, i.e., the users data set is not considered in the filtering phase and this may result in un-necessary I/O cost on facility R*-tree. Consider an extreme example where the users data set is empty. In such case, the filtering phase is not required at all. However, each algorithm runs the filtering phase normally (aiming to filter as much space as possible) which results in un-necessary I/O cost. This experiment indicates that an interesting future work is to develop an algorithm that filters the space by considering the locations of facilities as well as users which may result in a lower overall I/O cost.



(a) North America (b) Los Angeles

(c) California (d) Synthetic small (2K)

(e) Synthetic medium (200K) (f) Synthetic Large (20M)

Figure 13: Effect of k on I/O cost

4.4 Effect of k

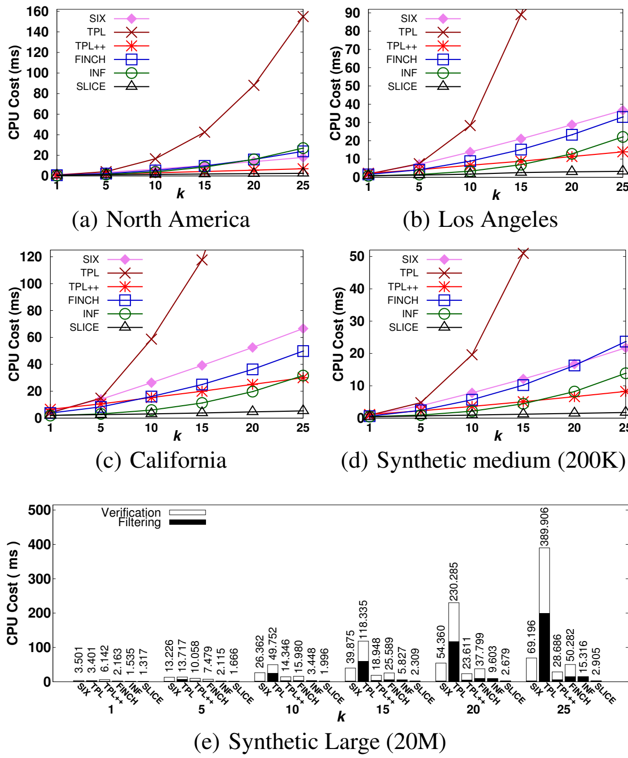
4.4.1 *I/O cost*

Figure 13 evaluates the I/O cost of the algorithms for varying k on real and synthetic data sets. TPL++ has the lowest I/O cost among all of the algorithms whereas SIX has the highest I/O cost. For larger data sets (e.g., Figure 13(c) and Figure 13(f)), the I/O cost of TPL++ and InfZone is almost the same. FINCH is the third best algorithm in terms of I/O cost followed by SLICE. Note that the effect of k for smaller data sets (e.g., Figure 13(d)) is more significant. This is mainly because the density of facilities is lower for smaller data sets which results in a more significant increase in the unfiltered area as the value of k increases.

4.4.2 *CPU time*

Figure 14 studies the effect of k on CPU cost of each algorithm. SLICE outperforms all other algorithms and scales quite well with the increase in k. This is due to the cheap filtering and verification strategies used by SLICE. InfZone is the second best algorithm for smaller values of k and TPL++ is the second best algorithm for larger values of k. Note that InfZone performs better (as compared to the other algorithms except SLICE) for larger data sets (e.g., synthetic large and California). This is mainly because InfZone and SLICE scale better as the data set size increases as we illustrate later in Figure 15.

The verification cost of TPL, TPL++, FINCH and SIX increases significantly with k (as can be seen in Figure 14(e)). This is because



(a) North America (b) Los Angeles

(c) California (d) Synthetic medium (200K)

(e) Synthetic Large (20M)

Figure 14: Effect of k on CPU time

these algorithms use range queries (or concurrent range queries) to verify the candidate users which is expensive. The number of candidates increases with the increase in k which results in larger verification cost. On the other hand, InfZone and SLICE use more efficient verification strategies and therefore the increase in verification cost for these algorithms is not significant.

The filtering cost of TPL, TPL++, FINCH and InfZone rapidly increases as the value of k increases. The effect of k on the filtering cost of TPL is more adverse than the other algorithms mainly because the filtering cost is much larger (e.g., $O(km)$ as compared to $O(m)$). The filtering cost of InfZone and FINCH is also significantly increased mainly because the construction of the influence zone and convex polygon depends on m which increases with the increase in k. Note that the filtering cost of SIX and SLICE does not significantly increase which is due to the cheap filtering rules used by these algorithms.

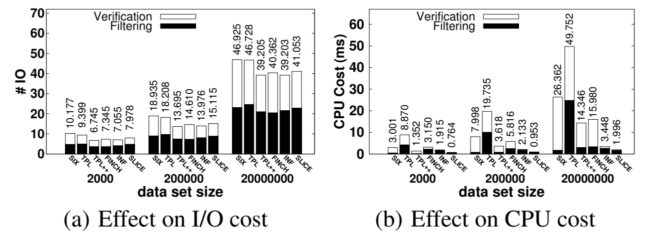
Note that TPL++ is slower than TPL for $k = 1$ in contrast to the results presented in Section 3.3. This is because of the optimization used in the implementation of TPL as explained in Section 4.1, i.e., `kTrim` is repeatedly called as long as the trimmed entry is smaller than the input entry. This improves the CPU and I/O cost of TPL.

4.5 Effect of data set size

In this section, we study the effect of the change in data set size on each algorithm. Similar to existing experimental studies [6, 28], we note that a *moderate* change in the data set size (e.g., a few times) does not have significant effect on the cost. We omit the results due to space limitations.

In Figure 15, we study the effect of *enormous* change on the data set size. Specifically, we vary the data set size from 2000 points to $20,000,000$ points and study the effect on I/O cost (Figure 15(a)) and CPU cost (Figure 15(b)). The I/O cost of each algorithm increases mainly because of the increase in the total number of nodes.

We remark that although the total number of users increases



(a) Effect on I/O cost (b) Effect on CPU cost

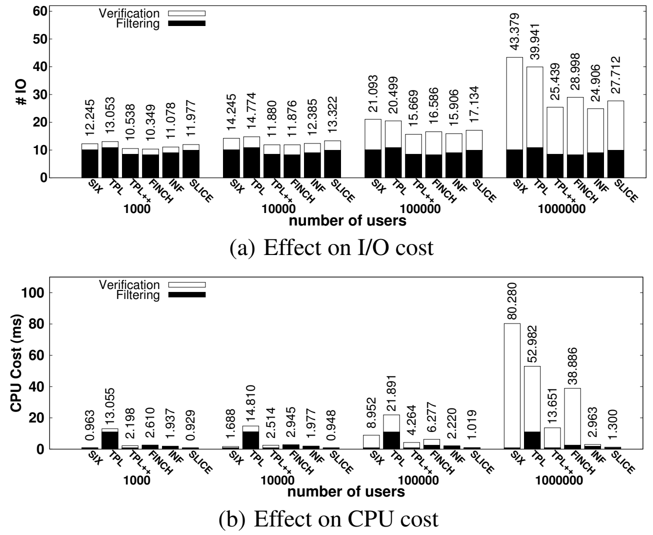
Figure 15: Effect of data set size (enormous change)

with the data size, the number of candidates does not necessarily increase. This is because the number (and density) of facilities also increases which results in a larger filtered area. Figure 15(b) shows that the verification cost of TPL, TPL++, FINCH and SIX increases. This is because the cost of issuing range queries on larger data sets increases significantly. On the other hand, the verification phase of InfZone and SLICE is cheaper because the verification cost mainly depends on m (number of filtering facilities) and m does not increase with the change in data set size.

The filtering cost of TPL increases significantly mainly because the number of entries checked for filtering increases significantly. Since the cost of checking whether an entry can be filtered or not is lower for the other algorithms, the filtering cost of these algorithms does not increase as significantly as that of TPL.

4.6 Effect of relative data size

In this section, we study the performance of different algorithms by varying $|U|/|F|$ where $|U|$ and $|F|$ are the total number of users and facilities, respectively. Specifically, in Figure 16, we fix the number of facilities to $100,000$ and vary the number of users whereas, in Figure 17, the number of users is fixed to $100,000$ and the number of facilities is changed.

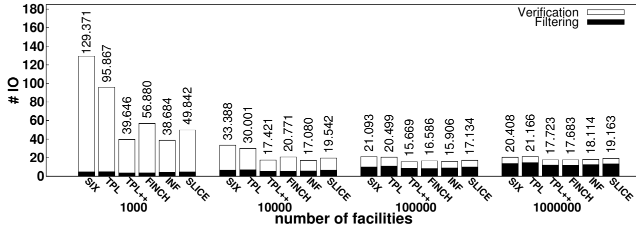


(a) Effect on I/O cost

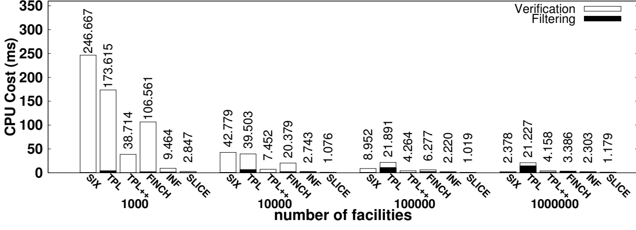
(b) Effect on CPU cost

Figure 16: Varying the number of users ($|F| = 100,000$)

Figure 16 studies the effect on the I/O cost and CPU cost of the algorithms. Since the number of facilities does not change, the filtering cost of each algorithm remains unchanged. The verification cost increases with the increase in the size of users data set because the number of unpruned nodes (and the number of candidates) increases. Figure 16(b) shows that InfZone and SLICE scale better than the other algorithms in terms of CPU cost. This is due to the cheaper verification used by these algorithms.



(a) Effect on I/O cost



(b) Effect on CPU cost

Figure 17: Varying the number of facilities ($|U| = 100,000$)

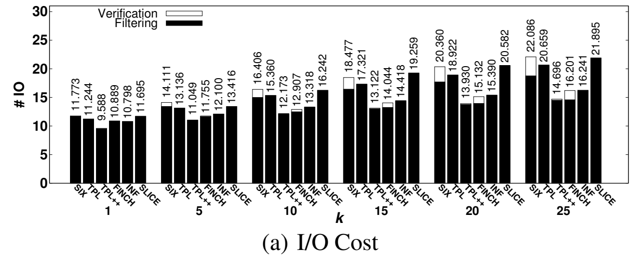
In Figure 17, we fix the number of users to 100,000 and study the effect of changing the number of facilities. As the number of facilities increases, the filtering cost of each algorithm increases. This is because the facility R*-tree is larger and the number of nodes that cannot be filtered increases. The verification cost of each algorithm decreases with the increase in the number of facilities. This is because the filtered area becomes larger when the density of the facilities data set is increased. This results in a fewer number of unfiltered nodes and candidate users. InfZone and SLICE are less sensitive to the change in number of facilities because the verification cost of these two algorithm is significantly lower than the other algorithms.

Figure 16 and Figure 17 show that the performance gain of InfZone as compared to other algorithms is high when $|U|/|F|$ is large and low when $|U|/|F|$ is small. This is because the main strength of InfZone is its cheap verification phase. If $|U|/|F|$ is small, the effect of its strength is smaller and vice versa.

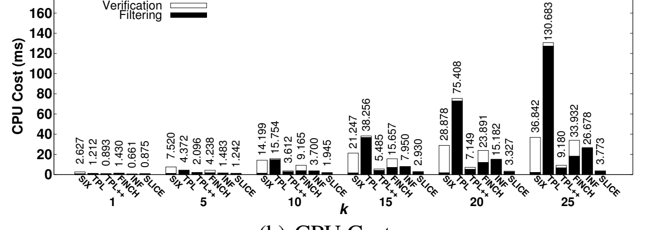
4.7 Monochromatic queries

Each of the algorithms can be easily applied to answer monochromatic RkNN queries. The major difference is that the candidates are the facilities found in S_{fil} after the filtering phase. These facilities are then verified as explained earlier in Section 3. For details, please see the original papers. Since TPL++ does not apply filtering to the facilities, its S_{fil} (i.e., the number of candidates) is larger. First, each of the candidate facilities is filtered using other facilities in S_{fil}. Then, the remaining facilities are verified using the concurrent range queries.

Figure 18 evaluates the algorithms for monochromatic RkNN queries on LA data set. Note that the filtering cost is the major I/O cost for each algorithm. This is because the algorithms do not need to access any node of the facility R*-tree to retrieve the candidate set. The I/O cost in the verification phase is the cost of range queries for TPL, TPL++, FINCH and SIX whereas the verification I/O cost of InfZone and SLICE is zero because these two algorithms access all facilities required for the verification during the filtering phase. The I/O cost of SLICE is similar to SIX and worse than all other algorithms. This is because SLICE tries to access all the facilities that are to be used in the verification phase. This results in accessing un-necessary nodes of the R*-tree. InfZone also aims at accessing all the facilities required during the verification but its I/O cost is lower than SLICE because it uses more powerful filtering strategy.

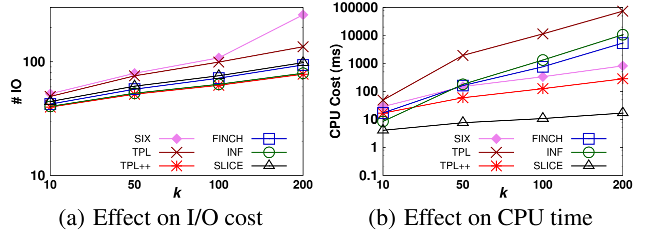


(a) I/O Cost



(b) CPU Cost

Figure 18: Monochromatic Queries (LA data set)



(a) Effect on I/O cost (b) Effect on CPU time

Figure 19: Effect of large k (CA data set)

Figure 18(b) shows the CPU cost for each algorithm. The results are similar to those for bichromatic RkNN queries. Specifically, SLICE outperforms other algorithms and scales better with k. TPL++ is the second best algorithm for larger values of k and InfZone is the second best algorithm for smaller values of k.

4.8 Effect of larger k

Figure 19 studies the effect of relatively larger k (up to 200) on each algorithm. Figure 19(a) shows that the I/O cost of the algorithms shows similar trend as for smaller values of k, e.g., TPL++ and InfZone are the best algorithms whereas SIX and TPL are the most expensive algorithms in terms of I/O cost. Figure 19(b) shows that SLICE scales much better than the other algorithms in terms of CPU cost. Please note log-scale is used in the figures. InfZone and TPL scale the worst among all algorithms. This is because TPL requires $O(km)$ to filter each entry where m significantly increases with k. The cost of influence zone increases mainly because updating the influence zone for each new facility in S_{fil} takes $O(m^2)$ and m increases with k. FINCH also requires $O(m^2)$ but it scales better than InfZone because the value for m for FINCH is smaller than that of InfZone and the difference increases with k.

5. CONCLUSION

To summarize our findings, Table 2 provides rankings of the algorithms under different experimental settings. Although the table is self-explanatory, we provide a few comments. In terms of I/O cost, TPL++ and InfZone are the best algorithms. Specifically, TPL++ generally performs better than InfZone when the facilities data set is larger or equal to the users data set. On the other hand,

Criteria	1st	2nd	3rd	4th	5th	6th
I/O Cost (no buffer)	{TPL++, InfZone}		SLICE	TPL	FINCH	SIX
I/O Cost (small buffer)	{TPL++, InfZone}		FINCH	SLICE	{TPL, SIX}	
CPU Cost ($k \leq 10$)	SLICE	InfZone	TPL++	FINCH	{SIX, TPL}	
CPU Cost ($10 < k \leq 25$)	SLICE	{InfZone, TPL++}		FINCH	SIX	TPL
CPU Cost ($25 < k \leq 200$)	SLICE	TPL++	SIX	FINCH	InfZone	TPL
Ease of implementation	{SIX, SLICE}		{TPL, TPL++}		{FINCH, InfZone}	

Table 2: Ranking of the algorithms under different criteria

InfZone performs better when the users data set is larger. In terms of CPU cost, SLICE is the best algorithm for all values of k. InfZone is the second best algorithm for smaller values of k whereas TPL++ is the second best algorithm for larger values of k.

We also rank the algorithms considering the ease of implementation. Note that this ranking is based on our personal experience and is subjective. SIX and SLICE are the easiest to implement because the half-space based filtering is generally more complicated to implement. FINCH and InfZone are more difficult to implement than TPL and TPL++ mainly because these algorithms require updating the polygons as well as the counters of intersection points of the half-spaces.

We also compared the I/O cost of the algorithms with the lower bound I/O cost. Our experimental study demonstrates that the number of I/Os on users R*-tree is quite close (or equal) to the lower bound for some of the algorithms. However, the number of I/Os on facilities R*-tree can be further improved especially for the data sets where the number of users is significantly smaller than the number of facilities, e.g., by devising filtering strategies that also consider the locations of users. We remark that the lower bound holds only for the case when data sets are indexed by R*-tree. The lower bound may be significantly smaller for specialized indexes.

Acknowledgment. We are sincerely thankful to the inventors of TPL [20] and FINCH [25] for sending us their source codes that helped us to confirm that our implementation of these algorithms does not have any glitches. Muhammad Aamir Cheema is supported by ARC DE130101002 and DP130103405. The research of Xuemin Lin is supported by NSFC61232006, NSFC61021004 and ARC (DP120104168, DP140103578, DP150102728). Wei Wang is supported by ARC DP130103401 and DP130103405.

6. REFERENCES

[1] Factors affecting direct attached storage device performance in the application layer. *HP Technical Brief*, 2012.
[2] E. Achtert, H.-P. Kriegel, P. Kröger, M. Renz, and A. Züfle. Reverse k-nearest neighbor search in dynamic and general metric databases. In *EDBT*, pages 886–897, 2009.
[3] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, pages 44–53, 2002.
[4] T. Bernecker, T. Emrich, H.-P. Kriegel, M. Renz, , and S. Z. A. Züfle. Efficient probabilistic reverse nearest neighbor query processing on uncertain data. *PVLDB*, 4(10):669–680, 2011.
[5] M. A. Cheema, X. Lin, W. Wang, W. Zhang, and J. Pei. Probabilistic reverse nearest neighbor queries on uncertain data. *IEEE Trans. Knowl. Data Eng.*, 22(4):550–564, 2010.
[6] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *ICDE*, pages 577–588, 2011.
[7] M. A. Cheema, X. Lin, Y. Zhang, W. Wang, and W. Zhang. Lazy updates: An efficient technique to continuously monitoring reverse knn. *PVLDB*, 2(1):1138–1149, 2009.
[8] M. A. Cheema, Z. Shen, X. Lin, and W. Zhang. A unified framework for efficiently processing ranking related queries. In *Proc. 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, March 24-28, 2014.*, pages 427–438, 2014.
[9] M. A. Cheema, W. Zhang, X. Lin, and Y. Zhang. Efficiently processing snapshot and continuous reverse k nearest neighbors queries. *VLDB J.*, 21(5):703–728, 2012.
[10] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li. Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. *VLDB J.*, 21(1):69–95, 2012.
[11] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. *PVLDB*, pages 291–302, 2007.
[12] J. Goldstein, R. Ramakrishnan, U. Shaft, and J.-B. Yu. Processing queries by linear constraints. In *PODS*, pages 257–267, 1997.
[13] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, pages 201–212, 2000.
[14] Y. Park, J.-K. Min, and K. Shim. Parallel computation of skyline and reverse skyline queries using mapreduce. *PVLDB*, 6(14):2002–2013, 2013.
[15] F. P. Preparata and M. I. Shamos. *Computational Geometry An Introduction*. Springer, 1985.
[16] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *USENIX Winter*, pages 405–420, 1993.
[17] M. Sharifzadeh and C. Shahabi. Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. *PVLDB*, 3(1):1231–1242, 2010.
[18] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop*, pages 44–53, 2000.
[19] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of influence sets in frequently updated databases. *PVLDB*, pages 99–108, 2001.
[20] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. *PVLDB*, pages 744–755, 2004.
[21] Y. Tao, D. Papadias, X. Lian, and X. Xiao. Multidimensional reverse *k* nn search. *VLDB J.*, 16(3):293–316, 2007.
[22] Y. Tao, M. L. Yiu, and N. Mamoulis. Reverse nearest neighbor search in metric spaces. *IEEE Trans. Knowl. Data Eng.*, 18(9):1239–1252, 2006.
[23] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD*, pages 59–72, 2009.
[24] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE*, pages 365–376, 2010.
[25] W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan. Finch: Evaluating reverse k-nearest-neighbor queries on location data. *PVLDB*, 1(1):1056–1067, 2008.
[26] www.census.gov/geo/maps-data/data/tiger line.html.
[27] www.cs.fsu.edu/%7Elifeifei/SpatialDataset.htm.
[28] S. Yang, M. A. Cheema, X. Lin, and Y. Zhang. Slice: Reviving regions-based pruning for reverse k nearest neighbors queries. In *ICDE*, pages 760–771, 2014.
[29] B. Yao, F. Li, and P. Kumar. Reverse furthest neighbors in spatial databases. In *ICDE*, pages 664–675, 2009.
[30] M. L. Yiu and N. Mamoulis. Reverse nearest neighbors search in ad hoc subspaces. *IEEE Trans. Knowl. Data Eng.*, 19(3):412–426, 2007.
[31] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. *IEEE Trans. Knowl. Data Eng.*, pages 540–553, 2006.
[32] A. W. Yu, N. Mamoulis, and H. Su. Reverse top-k search using random walk with restart. *PVLDB*, 7(5):401–412, 2014.