THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRC7/06

ERkNN: Efficient Reverse k-Nearest Neighbors Retrieval with Local kNN-Distance Estimation

Chenyi XIA, Wynne HSU and Mong Li LEE

July 2006

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

ERkNN: Efficient Reverse k-Nearest Neighbors Retrieval with Local kNN-Distance Estimation

Chenyi Xia
Department of Computer Science, National University of Singapore 117543
xia.chenyi@nus.edu.sg

Wynne Hsu
Department of Computer Science, National University of Singapore 117543
whsu@comp.nus.edu.sg

Mong Li Lee
Department of Computer Science, National University of Singapore 117543
leeml@comp.nus.edu.sg

ABSTRACT

The Reverse k-Nearest Neighbors (RkNN) queries are important in profile-based marketing, information retrieval, decision support and data mining systems. However, they are very expensive and existing algorithms are not scalable to queries in high dimensional spaces or of large values of k. This paper describes an efficient *estimation-based* RkNN search algorithm (ERkNN) which answers RkNN queries based on *local kNN-distance estimation* methods. The proposed approach utilizes estimation-based filtering strategy to lower the computation cost of RkNN queries. The results of extensive experiments on both synthetic and real life datasets demonstrate that ERkNN algorithm retrieves RkNN efficiently and is scalable with respect to data dimensionality, k, and data size.

Categories and Subject Descriptors: H.3.3 [INFORMATION STORAGE AND RETRIEVAL]: Information Search and Retrieval

General Terms: Algorithms, Performance, Design, Experimentation

1. INTRODUCTION

The reverse k-Nearest Neighbors (RkNN) query aims to find points in a data set that have the given query point as one of their k-nearest neighbors (kNN). It has many applications in profile-based marketing, information retrieval, decision support and data mining systems and therefore, has received considerable attention in the recent years[10, 15, 18, 14, 9, 11].

RkNN queries are much more complex than the traditional kNN queries because unlike kNN, RkNN are not localized to the neighborhood of the query point. We illustrate this behavior using the example in Figure 1 where p_2 is the query point and k=2. We observe that p_2 is one of the 2-nearest neighbors of p_1, p_3, and p_4. Hence, p_2's reverse 2-nearest neighbors are p_1, p_3, and p_4. Note that p_4 is an R2NN of p_2 although it is far from the query point p_2. In contrast, p_5 and p_7 are not answers of the R2NN query of p_2 although they are close to p_2.

The naive solution for RkNN search is expensive. It first computes the k-nearest neighbors for each point p in the dataset and retrieves points that have q as one of the k-nearest neighbors. The complexity of this naive approach is $O(N^2)$ for non-indexed data, and $O(NlogN)$ for datasets that are indexed by some hierarchical structure such as the R-tree[4] (N is the cardinality of dataset).

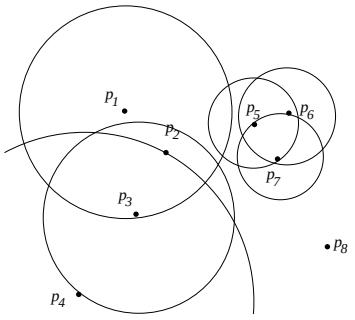


Figure 1: An RkNN query example ($k = 2$).

Various methods have been developed for the efficient processing of RkNN queries and can be divided into two categories: *pre-computation* and *space pruning*. *Pre-computation* methods [10, 18] pre-compute the nearest neighbors of each point in the datasets and store the pre-computed information in hierarchical structures. This approach cannot answer an RkNN query unless the corresponding k-nearest neighbor information is available. *Space pruning* methods such as [13, 16, 14] utilize the geometry properties of RNN to find a small number of data points as candidates and then verify them with NN queries or range queries. However, these methods are expensive when data dimensionality is high or when the value k is large.

This paper presents an efficient *estimation-based* RkNN search algorithm called ERkNN utilizes the filter-and-refine framework. It retrieves RkNN candidates based on the estimated kNN-distance (kNN-distance is the distance from a data point to its kth-nearest neighbor). This estimation-based filter has the advantage that its computation cost is much lower than the filtering strategies employed by space pruning methods. This improves the RkNN query speed by orders of magnitude. We provide two local kNN-distance estimation methods - the PDE method and the kDE method. Extensive experiments on both synthetic and real-world datasets demonstrate that ERkNN retrieves RkNN efficiently and is scalable with respect to data dimensionality, k, and data size.

The rest of the paper is organized as follows. Section 2 defines the problem and reviews related work. Section 3 describes the estimation-based RkNN search algorithm. Section 4 presents the results of the performance study, and we conclude in Section 5.

CIKM'05, October 31–November 5, 2005, Bremen, Germany.

2. PRELIMINARY

2.1 Problem Statement

We focus on the conventional RkNN query [10] in our study. It is formally defined as follows.

Definition 2.1 (Reverse k-Nearest Neighbors) *Given a dataset DB, a query point q, a positive integer k and a distance metric $\ell()$, reverse k-nearest neighbors of q, denoted as $RkNN(q)$, is a set of points $S \subseteq DB$ such that $\forall p \in S, q \in kNN(p)$, where $kNN(p)$ are the k-nearest neighbors of point p.*

Data points in our study are multi-dimensional vectors $p =< x_1, x_2, ..., x_d >$. They could be locations and features vectors of documents and images etc. The distance metric in our consideration is the L_ρ metric, where $\ell(p,q) = \sum_{i=1}^{d} |p.x_i - q.x_i|^\rho \;^{1/\rho}$, $1 \leq \rho \leq \infty$. For illustrative purposes, we shall use the most commonly used metric, L_2 (the Euclidean distance). Table 1 summarizes the symbols used frequently in the paper.

2.2 Related Work

Various methods have been proposed for efficient RkNN query processing. The majority have been designed for RNN queries (i.e., RkNN queries when $k = 1$). We divide these methods into two categories: *pre-computation* methods and *space pruning* methods.

Pre-computation methods [10, 18] pre-compute and store the nearest neighbor information of each point in a dataset in index structures, i.e., the RNN-tree [10] and the Rdnn-tree[18]. With the pre-computed nearest neighbor information, an RNN query is answered by a *point enclosure query* [10] that retrieves a set of points $\mathcal{A}$ such that for each point $p \in \mathcal{A}$, the query point falls within the sphere centered at p and of the radius dnn (the distance from p to its nearest neighbor). The RNN-tree and the Rdnn-tree can be used to answer RkNN queries by pre-computing and storing the k-nearest neighbor information in these structures and applying the same point enclosure search.

The drawback of *pre-computation* methods is that they cannot answer an RkNN query unless the corresponding k-nearest neighbor information is available. Since the values of k may vary greatly in many applications, storing the k-nearest neighbor information for all possible values of k is expensive and sometimes infeasible, and maintaining such a large amount of k-nearest neighbor information in the presence of frequent updates is even more costly.

Space pruning methods such as TPL [16], SFT [13], SAA [14] filters a large portion of data points based on the geometry properties of RkNN and returns a small number RkNN candidate points for verification. SAA [14] makes use of the *bounded output* property, e.g. for an RNN query in the 2-dimensional space, a query point q has at most 6 RNNs [14]. Thus, SAA divides the data space into six equal regions by straight lines that intersect at the query point q. For each region, the nearest neighbors of q are retrieved as candidates and then verified by NN queries. For an RkNN query, the number of outputs is bounded by $6 \cdot k$ in 2-dimensional space [16] and similar algorithm is applicable to RkNN queries. SAA is costly for high-dimensional data because the bounding number increases exponentially with respect to data dimensionality.

SFT [13] is based on the assumption that RkNN and KNN are correlated, that is, an RkNN of q is expected to be one KNN of q, where K is a value bigger than k. It retrieves K nearest points to q as candidates and then verifies the candidates with range queries. However, the correlation between RkNN and KNN is not strong. Hence, K should be set to be sufficiently big in order to reduce *false misses* (points that are RkNN but missed from the found answer set), which makes it expensive for ks of large values.

Symbol	Definition
d	data dimensionality
k or $\mathcal{K}$	an integer, number of nearest neighbors
p and q	data point and query point
$\ell(p,q)$	distance between points p and q
$dnn_k(p)$	kNN-distance - distance between p and its kth-nearest neighbor
$ednn_k(p)$	estimated kNN-distance
$dnn_{\mathcal{K}}(p)$	$\mathcal{K}$NN-distance - distance between p and its $\mathcal{K}$th-nearest neighbor

Table 1: Symbols and definitions.

TPL [16] uses the *half-planes* pruning strategy to divide the data space into two half-planes by the perpendicular bisector between q and an arbitrary data point p. Then any point in the half plane of p cannot be an RNN of q. TPL traverse the R-tree to retrieve nearest neighbors incrementally as RkNN candidates and uses the candidates to prune tree nodes with the *trim* algorithm until all nodes of R-tree are either pruned or visited. The retrieved candidates are verified by an I/O optimized refinement algorithm. For RkNN queries, the *k-trim* algorithm is used to prune R-tree nodes based on the extended *half-planes* strategy. TPL is efficient in low-dimensional space and for small values of k. However, the cost of *k-trim* algorithm increases rapidly with respect to k.

3. ESTIMATION-BASED RKNN SEARCH

Algorithm 1 shows the estimation-based RkNN search (ERkNN) which has two main steps. The first step calls procedure Filter to retrieve a set of points p whose distance to the query point q is equal to or greater than p's estimated kNN-distance as RkNN candidates. The second step calls procedure Refinement to verify the candidates with range queries.

The novelty of ERkNN lies in its efficient candidate retrieval based on the *local kNN-distance estimation* which accurately approximates each point's kNN-distance. It answers RkNN queries of arbitrary k efficiently without requiring the corresponding k-nearest neighbor information and outperforms *space pruning* methods significantly especially when data dimensionality is high and k is big. The following subsections give the details of the estimation methods and each procedure.

Algorithm 1 ERkNN(T, q, k)

Input:
T is the index tree, q is the query point, k is an integer.
Output:
RkNN answers.
Description:
1: $\mathcal{A} = \emptyset$; /*$\mathcal{A}$ is the RkNN candidate set*/
2: **Filter**(T, q, k, $\mathcal{A}$);
3: **Refinement**(T, q, k, $\mathcal{A}$);

3.1 Local kNN-Distance Estimation Methods

Previous studies on the kNN-distance estimation [3] employ the *global uniform assumption*, that is, the data are uniformly distributed over the whole data space. We call these approaches the *global kNN-distance estimation*. The kNN-distance computed by the *global kNN-distance estimation* is the *average* of the kNN-distance of all the points in the dataset. However, the local density of each point in a dataset varies considerately and so does the kNN-distance of each point. RkNN candidate retrieval that is based on the average kNN-distance tends to produce a large number of false misses and false hits, which increases the refinement cost and as well as decreases the recall of the correct RkNN answers.

In this work, we introduce the idea of the *local kNN-distance estimation* which is based on the nonparametric density estima-

tion [7]. The nonparametric density estimation estimates a point's local density function by a small number of neighboring samples around the point. The resulting local density function gives a much better approximation of the data distribution compared to the global uniform assumption. Hence, the *local kNN-distance estimation* approximates the kNN-distance of each point much more accurately than the global approach.

We develop two *local kNN-distance estimation* methods - the PDE method and the kDE method. The PDE method is based on the parzen density estimator with uniform kernel [7], which is a most commonly-used non-parametric density estimator. The kDE method, as an alternative to the PDE method, is based on the interesting finding which is deduced from the kNN density estimator [7]. Our experiment study show that these methods produce similar estimation results, work effectively on both synthetic and real life datasets and outperform the global approach significantly.

3.1.1 The PDE method

The PDE method is based on the parzen density estimator with uniform kernel [7]. Let $L(p)$ be a small sphere region centered at p. The Parzen Density Estimator counts the number of points falling in $L(p)$ and estimates the local probability density function at p, $\hat{\mathrm{X}}(p)$ as follows:

$$\hat{\mathrm{X}}(p) = \frac{k/N}{V} \tag{1}$$

where N is the cardinality of dataset, k is the number of points within $L(p)$ and V is the volume of $L(p)$. $L(p)$ is a d-dimensional hyper-sphere of radius $r = dnn_k(p)$, so

$$V = V_{d,r}(p) = \frac{\sqrt{\pi^d} \cdot r^d}{\Gamma(d/2+1)} = \frac{\sqrt{\pi^d} \cdot dnn_k(p)^d}{\Gamma(d/2+1)} \tag{2}$$

where, $\Gamma(x+1) = x\Gamma(x)$, $\Gamma(1) = 1$, $\Gamma(1/2) = \sqrt{\pi}$. Combining Equation 1 and 2, we have

$$\hat{\mathrm{X}}(p) = \frac{k/N \cdot \Gamma(d/2+1)}{\sqrt{\pi^d} \cdot dnn_k(p)^d} \tag{3}$$

When knowing that p's $\mathcal{K}$NN-distance is $dnn_{\mathcal{K}}(p)$, we have

$$\hat{\mathrm{X}}(p) = \frac{\mathcal{K}/N \cdot \Gamma(d/2+1)}{\sqrt{\pi^d} \cdot dnn_{\mathcal{K}}(p)^d} \tag{4}$$

Equation 4 gives an approximation of local density around p. From Equation 3, we also have

$$dnn_k(p) = \sqrt[d]{\frac{k \cdot \Gamma(d/2+1)}{N \cdot \sqrt{\pi^d} \cdot \hat{\mathrm{X}}(p)}} \tag{5}$$

Combining Equation 4 and 5 and applying the assumption that the data density is uniform over p's kNN vicinity to $\mathcal{K}$NN vicinity (the *local uniform assumption*), we obtain

$$dnn_k(p) = dnn_{\mathcal{K}}(p) \cdot \sqrt[d]{\frac{k}{\mathcal{K}}} \tag{6}$$

Thus, we have the PDE method which estimates kNN-distance of p using the following equation:

$$ednn_k(p) = dnn_{\mathcal{K}}(p) \cdot \sqrt[d]{\frac{k}{\mathcal{K}}} \tag{7}$$

where $ednn_k(p)$ is the estimated kNN-distance of p and $dnn_{\mathcal{K}}(p)$ is $\mathcal{K}$NN-distance, the distance between p and its $\mathcal{K}$th nearest neighbor, which is pre-computed in advance. d is data dimensionality.

3.1.2 The kDE method

The kDE method is based on an interesting finding which is deduced from the kNN density estimator [7, 8] [1], that is, the ratio of (k+1)NN-distance to the kNN-distance is as follows [7, 8]:

$$\frac{dnn_{k+1}}{dnn_k} \cong 1 + \frac{1}{kd}$$

Thus, we have,

[1]Refer [7] for the detail of deduction

Algorithm 2 Filter(T, q, k,$\mathcal{A}$)

Input:
T is the Rdnn-tree, q is the query point, k is an integer, $\mathcal{A}$ is the set of RkNN candidates.

Description:
```
1: Initialize queue Q with root of T;
2: while Q is not empty do
3:   Dequeue a node N from Q;
4:   if N is an internal node then
5:     for each sub-node N' of N do
6:       if MinDist(N', q) ≤ Max_ED(N') then
7:         Insert N' to Q;
8:   else
9:     for each point p in N do
10:      if ℓ(p, q) ≤ ednn_k(p) then
11:        Insert p into A;
```

$$\begin{aligned} dnn_{k+1} &\cong dnn_k \cdot (1 + \tfrac{1}{kd}) \\ dnn_{k+2} &\cong dnn_{k+1} \cdot \ 1 + \tfrac{1}{(k+1)d} \\ &\cong dnn_k \cdot \ 1 + \tfrac{1}{kd} \ \cdot \ 1 + \tfrac{1}{(k+1)d} \\ &\dots \end{aligned}$$

So for any two integers k_1 and k_2 ($k_1 \neq k_2$),

$$\begin{aligned} if \quad k_1 < k_2 \quad dnn_{k_2} &\cong dnn_{k_1} \cdot \textstyle\prod_{i=k_1}^{k_2-1}(1 + \frac{1}{i \cdot d}) \\ if \quad k_1 > k_2 \quad dnn_{k_2} &\cong \frac{dnn_{k_1}}{\prod_{i=k_2}^{k_1-1}(1+\frac{1}{i \cdot d})} \end{aligned}$$

Therefore, we have the kDE method which estimates kNN-distance using Equation 8:

$$ednn_k(p) = \begin{cases} dnn_{\mathcal{K}}(p) \cdot \prod_{i=\mathcal{K}}^{k-1}(1 + \frac{1}{i \cdot d}) & \text{if } k > \mathcal{K} \\ dnn_{\mathcal{K}}(p) & \text{if } k = \mathcal{K} \\ \frac{dnn_{\mathcal{K}}(p)}{\prod_{i=k}^{\mathcal{K}-1} 1+\frac{1}{i \cdot d}} & \text{if } k < \mathcal{K} \end{cases} \tag{8}$$

3.1.3 Discussion

The kNN-distance estimated by the PDE or the kDE methods is an approximation of the real kNN-distance, so the candidate set retrieved by the filter procedure of ERkNN may contain false hits and miss true answers due to the estimation error. The false hits will be removed with the refinement procedure of ERkNN. The problem of false misses will be discussed in Section 3.3.

For RkNN queries of different k values, ERkNN uses the same $\mathcal{K}$NN-distance as the basic for estimation. It is observed that when k is far from $\mathcal{K}$, the approximation becomes less accurate. This problem can be alleviated by a multiple $\mathcal{K}$s version of ERkNN. That is, we store several $\mathcal{K}$NN-distances ($\mathcal{K}_1$NN-distance, $\mathcal{K}_2$NN-distance,... $\mathcal{K}_m$NN-distance) and estimate a point's kNN-distance according to the $\mathcal{K}_i$NN-distance such that $\mathcal{K}_i$ is closest to k. ERkNN with multiple $\mathcal{K}$s is a straightforward extension of the single $\mathcal{K}$ case, so we will focus on the single $\mathcal{K}$ version ERkNN here.

The data dimensionality d can be evaluated by either the *embedded* dimensionality or the *intrinsic* dimensionality [17]. The *embedded* dimensionality is the length of the feature vector of data and the *intrinsic* dimensionality is the number of the *effective* features of data. Studies in query cost analysis and pattern recognition show that cost estimation and data analysis based on intrinsic dimensionality are more accurate. This is same for the *local kNN-distance estimation* according to our experimental study. The PDE and kDE methods estimate the kNN-distance more accurately when the *intrinsic* dimensionality is used in Equation 7 and 8. Approaches for intrinsic dimensionality computation are in [17].

3.2 Algorithm

We now present the filter and refinement procedures of ERkNN. We use the Rdnn-tree [18] data structure for the search.

Data Structure: The Rdnn-tree is basically an R-tree that is augmented with the nearest neighbor *distance* (NN-distance). We

Algorithm 3 Refinement(T, q, k, $\mathcal{A}$)

Input:
T is the Rdnn-tree, q is the query point, k is an integer, $\mathcal{A}$ is the set of RkNN candidates.

Description:
1: Initiate range queries;
2: **Refine_in_memory**($\Re$, $\mathcal{A}$);
3: **Range_Queries**(T, k, $\Re$, R_a, $\mathcal{A}$);
4: Output points p_i in $\mathcal{A}$;

store the data points and the $\mathcal{K}$NN-distance in the leaf nodes of the Rdnn-tree. Each leaf node entry e has the form (p, $dnn_{\mathcal{K}}(p)$), where p is the data point and $dnn_{\mathcal{K}}(p)$ is the $\mathcal{K}$NN-distance of p.

Each entry e in the *internal* nodes of the Rdnn-tree has the form (ptr, $MaxDnn_{\mathcal{K}}$, mbr). ptr points to a sub-node N'; mbr is the minimum bounding rectangle (MBR) of N'; $MaxDnn_{\mathcal{K}}$ is the maximal $\mathcal{K}$NN-distance of all data points in the subtree rooted at N'.

$$MaxDnn_{\mathcal{K}} = Max_{i=1}^{m} dnn_{\mathcal{K}}(p_i) \tag{9}$$

where $p_1, ..., p_m$ are all points within N'. We use the algorithm described in [18] to build the Rdnn-tree with $\mathcal{K}$NN-distance, except that the NN queries are replaced by the $\mathcal{K}$NN queries. The tree can also be constructed using the bulk approach proposed in [12].

3.2.1 Filter Procedure

In the filtering step, ERkNN retrieves a set of points p whose estimated kNN-distance is equal to or greater than distance from p to the query point q. The estimated kNN-distance $ednn_k(p)$ is computed with either the PDE or the kDE method. This estimation-based filter has the advantage that its computation cost is much lower than the filtering strategies employed by space pruning methods [15, 16, 13].

During the tree traversal, we apply the following *pruning strategy*: If $MinDist(N, q) \geq Max\_ED(N)$, tree node N can be pruned from traversal, where $MinDist(N, q)$ is the minimum distance between the query point q and the MBR of N and $Max\_ED(N)$ is computed as follows:

- The PDE method is employed for kNN-distance estimation:

$$Max\_ED(N) = \quad MaxDnn_{\mathcal{K}} \cdot \sqrt[d]{\frac{k}{\mathcal{K}}}$$

-The kDE method is employed for kNN-distance estimation:

$$Max\_ED(N) = \begin{cases} MaxDnn_{\mathcal{K}} \cdot \prod_{i=\mathcal{K}}^{k-1}(1 + \frac{1}{i \cdot d}) & \text{if } k > \mathcal{K} \\ MaxDnn_{\mathcal{K}} & \text{if } k = \mathcal{K} \\ \frac{MaxDnn_{\mathcal{K}})}{\prod_{i=k}^{\mathcal{K}-1} 1 + \frac{1}{i \cdot d}} & \text{if } k < \mathcal{K} \end{cases}$$

Since $MaxDnn_{\mathcal{K}} = Max_{i=1}^{m} dnn_{\mathcal{K}}(p_i)$, according to the computation of $Max\_ED(N)$, $Max\_ED(N) = Max_{i=1}^{m} ednn_k(p_i)$, where $ednn_k(p_i)$ is the estimated kNN-distance of p_i and p_i is a point in N. Therefore, a node N such that $MinDist(N, q) > Max\_ED(N)$ can be pruned from traversal. Algorithm 2 presents the candidate retrieval procedure that traverses the Rdnn-tree in a breath-first manner. It utilizes a queue Q to store tree nodes that shall be visited. Q contains the root of the Rdnn-tree initially. While Q is not empty, the algorithm dequeues a node N from Q and processes it according to the node type:

- If N is an *internal* node (line 4-7): for each sub-node N' represented by an entry e in N, it calculates $MinDist$ from the query point q to N', computes $Max\_ED(N')$ and inserts N' such that $MinDist(N', q) \leq Max\_ED(N')$ into Q to be visited later.
- If N is an *leaf* node (line 8-11): for each point p in N, it computes the distance between p and the query point q, estimates the kNN-distance of p and inserts points such that $\ell(p, q) \leq ednn_k(p))$ into the candidate set $\mathcal{A}$.

The algorithm stops when Q is empty, that is, all the tree nodes have been either visited or pruned. All the data points p such that $\ell(p, q) \leq ednn_k(p)$ are retrieved and stored in the candidate set $\mathcal{A}$.

The filter procedure of ERkNN is equal to the point enclosure query, so its complexity is $O(logN)$ [10], where N is the number of data points.

3.2.2 Refinement Procedure

The candidate set $\mathcal{A}$ contains false hits due to the over-estimation of a point p's kNN-distance. A refinement step is needed to remove the false hits.

A point p is a reverse k-nearest neighbor of q if and only if there are *less than* k points p' such that $\ell(p', p) < \ell(p, q)$ [13, 16]. According to this property, the refinement procedure removes candidates with a set of range queries. These range queries have the candidate points as the query points and the distances between the candidate points and the query point of the RkNN query q as query ranges. Candidates that have at least k points within their corresponding query ranges shall be removed from $\mathcal{A}$.

Algorithm 3 shows the four steps in the refinement procedure.

Step 1: Initialization of queries: for each point p_i in $\mathcal{A}$, a range query $R_i(p_i, r_i)$ is initialized and inserted into the query set $\Re$, where p_i is the query point, r_i is the query range and $r_i = \ell(p_i, q)$.

Step 2: A fast refinement in memory: the range queries are first evaluated among the candidates. That is, for each range query R_i, it checks how many candidate points are within R_i's query range. Candidate p_i that has at least k points within its query ranges is removed from $\mathcal{A}$.

Step 3: Range queries: it performs the range queries on the Rdnn-tree and removes data points that has at least k points in their query ranges from the candidate set.

Step 4: Points remains in $\mathcal{A}$ are output as RkNNs.

Steps 1-2 and 4 are straightforward. Step 3 dominates the cost incurred in the refinement procedure. In order to reduce both I/O and CPU cost, we apply the *aggregation strategy* in this step. The basic idea is to first compute an *aggregated range query* $R_a(p_a, r_a)$ before carrying out the individual range queries. The query range of R_a, which is centered at p_a and of radius r_a, covers the search ranges of all R_i in $\Re$. Figure 2 gives an example. There are three candidates p_1, p_2 and p_3. The dashed circles are their query ranges. The solid circle is the aggregated query R_a. The computation query sphere of R_a is corresponding to the *minimum enclosing ball* problems [6] whose complexity is lower bounded by $O(|\mathcal{A}|)$, where $|\cdot|$ is the cardinality of a set.

We design the following *pruning strategies* based on the aggregated query R_a:

Node pruning: For a node N, if $MinDist(N, p_a) \geq r_a$, N is surely out of the query range of any R_i in $\Re$ and can be cut off safely (*e.g.* N_1 in Figure 2). If $MinDist(N, p_a) < r_a$, we then check whether N intersects with at least one range query R_i in $\Re$. If N intersects with none of them, N can also be pruned away (*e.g.* N_2 in Figure 2).

Point pruning: For a data point p, if $\ell(p, p_a) \geq r_a$, p is surely out of the query range of any R_i in $\Re$. (*e.g.* p in Figure 2).

Query pruning: When a node N is being visited, if $MinDist(N, p_i) \geq r_i$, all the entries in N are surely out of the query range of R_i. Thus, R_i is marked *ignored* while N is being visited (*e.g.*, when N_3 in Figure 2 is being visited, R_1 and R_2 are to be ignored).

The above pruning strategies show that with the aggregated query R_a, a point p or a node N can be pruned away with a single distance computation of $\ell(p, p_a)$ or $MinDist(N, p_a)$ instead of checking its distance to each query point p_i. This saves a large amount of

Algorithm 4 Range_Queries(T, k, $\Re$, R_a, $\mathcal{A}$)

Input:
T is the Rdnn-tree, k is an integer, $\Re$ is a set of range query, R_a is the aggregated query of $\Re$.

Description:
1: Initialize c_i=0 for each query R_i in $\Re$;
2: Initialize priority queue Q with root of T;
3: **while** Q is not empty and $\Re$ is not empty **do**
4: Dequeue a node N from Q;
5: Apply *query pruning*;
6: **if** N is an internal node **then**
7: **for each** sub-node N' of N **do**
8: Apply *node pruning*;
9: Insert N' into Q if it cannot be pruned;
10: **else**
11: **for each** point p in N **do**
12: **if** p cannot be pruned by *point pruning* **then**
13: **for** each *not ignored* R_i in $\Re$ **do**
14: **if** $\ell(p, p_i) < r_i$ **then**
15: Increase c_i by 1;
16: **if** $c_i = k$ **then**
17: Remove R_i from $\Re$ and p_i from $\mathcal{A}$;

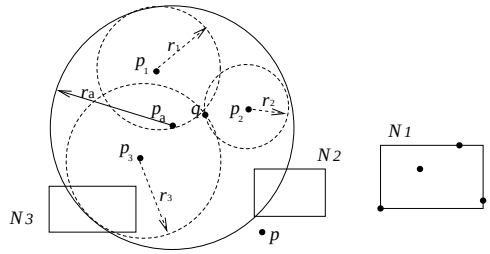


Figure 2: Query aggregation and illustration of pruning.

distance computation and reduces the CPU cost.

Algorithm 4 describes the procedure Range_Queries. c_i counts the number of points within query range of R_i. Q is a priority queue and sorts nodes in ascending order of their $MinDist$ to p_a. Initially, Q contains the tree root. When the first queue item N is dequeued, the query pruning strategy is applied to mark the queries such that $MinDist(N, p_i) \geq r_i$ as *ignored* (line 5). Node N is then processed according to its type:

- If N is an *internal* node (line 6-9): for each sub-node N' represented by an entry e in N, it applies the *node pruning* strategies. Node N' that cannot be pruned are inserted into the priority queue Q and shall be visited later.
- If N is an *leaf* node (line 10-17): for each point p in N, it first applies the *point pruning* strategy. If p is not pruned, it checks whether p is within the query range of each *not ignored* query R_i. If true, c_i is increased by 1. Whenever there are k points inside of query range of R_i, p_i is identified as a false hit and removed from $\mathcal{A}$ and range query R_i is also removed from $\Re$.

The procedure stops when either Q is empty or $\Re$ is empty, implying that all the tree nodes that intersect with at least one range query R_i in $\Re$ have been searched or the RkNN query has an empty answer set (*e.g.*, the R2NN of p_8 in Figure 1 is empty).

The I/O cost of the refinement procedure is $O(logN)$ since it carries out a set of range queries simultaneously, where N is the number of data points. The CPU cost is $O(|\mathcal{A}| \cdot logN)$. The upper bound of the I/O and CPU costs are $O(N)$ and $O(|\mathcal{A}| \cdot N)$ respectively even when the index fails.

3.3 Analysis of Recall

ERkNN may miss some correct answers due to the estimation error. This section examines the *recall* of the RkNN answer set retrieved by ERkNN.

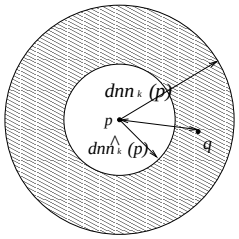


Figure 3: Points within the shade area are false misses.

Definition 3.1 *Let $\mathcal{A}$ be the RkNN answer set retrieved by ERkNN and Ω be the complete answer set of the RkNN query, the* recall *of $\mathcal{A}$ is denoted as $\mathcal{R}_{\mathcal{A}}=\frac{|\mathcal{A}|}{|\Omega|}$, where $|\cdot|$ is the cardinality of a set.*

Theorem 3.1 *The recall of the answer set retrieved by ERkNN is lower-bounded by $\int_0^\infty f(x)dx$, where $f(x)$ is the probability distribution of the estimation errors.*

Proof: For a point p in the complete answer set Ω, p is falsely missed when both the following conditions are true: (1) $ednn_k(p) < dnn_k(p)$; (2) $ednn_k(p) < \ell(p, q)$ (see Figure 3 as an illustration). Let $Pr\{\cdot\}$ be the probability of an event.

$$\mathcal{R}_{\mathcal{A}} = 1 - \frac{|\ \Omega\ | \cdot Pr\{ednn_k(p) < \ell(p,q) \cap ednn_k(p) < dnn_k(p)\}}{|\ \Omega\ |}$$

$$= 1 - Pr\{ednn_k(p) < \ell(p,q) \cap ednn_k(p) < dnn_k(p)\}$$

Since $Pr\{ednn_k(p) < \ell(p,q) \cap ednn_k(p) < dnn_k(p)\}$ $< Pr\{ednn_k(p) < dnn_k(p)\}$, $\mathcal{R}_{\mathcal{A}} \geq 1 - Pr\{ednn_k(p) < dnn_k(p)\}$ $= 1 - Pr\{ednn_k(p) - dnn_k(p) < 0\}$.

$ednn_k(p) - dnn_k(p)$ is the estimation error. Let $err(p) = ednn_k(p) - dnn_k(p)$, and $f(x)$ be the probability distribution of $err(p)$.

$$\mathcal{R}_{\mathcal{A}} \geq 1 - Pr\{err(p) < 0\} = 1 - \int_{-\infty}^{0} f(x)dx = \int_0^\infty f(x)dx$$

Hence, we have Theorem 3.1.

The above study shows that the false misses are introduced by the under-estimation of the kNN-distance. Therefore, we can improve the recall by simply introducing a positive adjustment to the estimated kNN-distance. Here, we present two approaches, the *local* adjustment and the *global* adjustment. Let $ednn_k{}'(p)$ be p's estimated kNN-distance after adjustment.

Local adjustment: Let ξ be a real number and $\xi > 1$.

$$ednn_k{}'(p) = ednn_k(p) \cdot \xi \tag{10}$$

ξ is called the local adjustment factor.

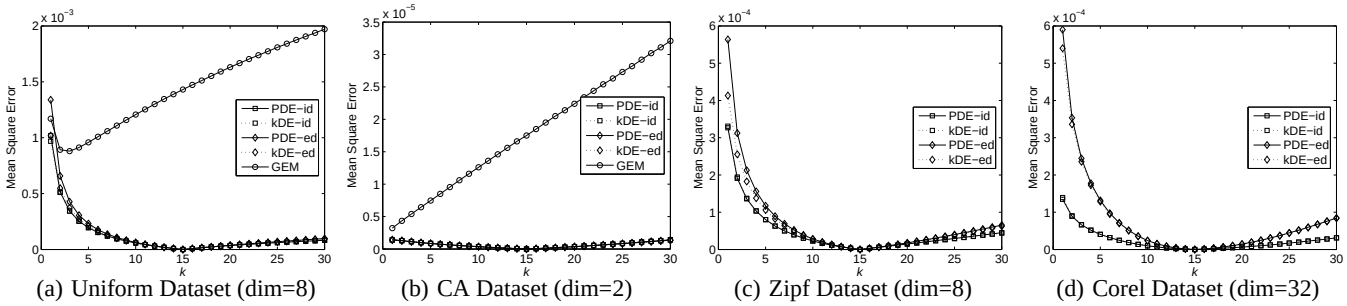
Global adjustment: Let λ be a real number and $\lambda > 0$.

$$ednn_k{}'(p) = ednn_k(p) + \lambda \tag{11}$$

λ is called the global adjustment factor.

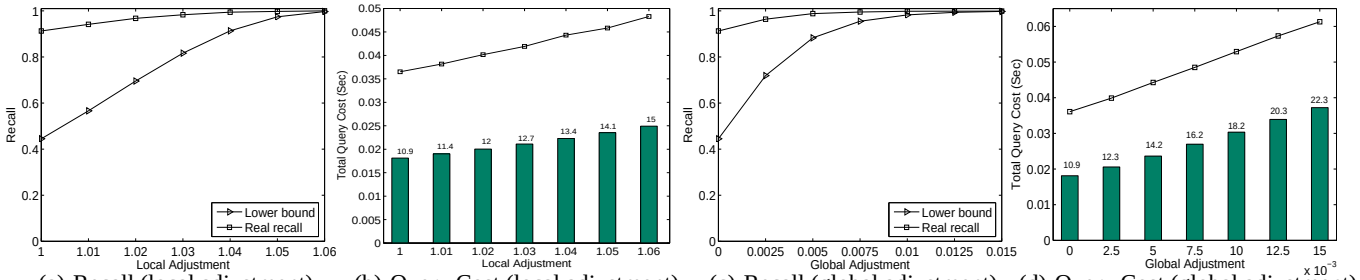
$ednn_k(p)$ in both Equation 10 and 11 shall be substituted with either Equation 7 or 8. ERkNN then retrieves a set of points p such that $\ell(p, q) \leq ednn_k{}'(p)$ in the filtering phase and then apply the same refinement algorithm to verify the candidates.

These adjustments reduces $Pr\{err(p) < 0\}$ and hence increases the recall. At the same time, the adjustment makes the filtering step of ERkNN retrieve more data points as candidates which may increase the refinement cost. However, the in-memory refinement procedure filters a number of candidates effectively, so its impact on the overall performance is not significant.



(a) Uniform Dataset (dim=8) (b) CA Dataset (dim=2) (c) Zipf Dataset (dim=8) (d) Corel Dataset (dim=32)

Figure 4: Comparison of kNN-distance Estimation Methods



(a) Recall (local adjustment) (b) Query Cost (local adjustment) (c) Recall (global adjustment) (d) Query Cost (global adjustment)

Figure 5: Study of ERkNN with Adjustment

4. PERFORMANCE STUDY

In this section, we present the results of our experiments to evaluate ERkNN. We use both synthetic and real life datasets. The size of the synthetic datasets ranges from 100K to 500K with dimensions ranging from 2 to 32. The real life datasets are the Corel dataset from UCI KDD data repository [1] which contains 32 dimensional feature vectors of around 60K images, and the CA dataset from the Sequoia 2000 benchmark [2] which contains 62,556 locations in California.

We compare ERkNN with TPL [16] and SFT [13]. We exclude SAA because its performance is significantly worse than SFT and TPL [16]. For both the R-tree (used by SFT and TPL) and the Rdnn-tree (used by ERkNN), the node size is 8192 bytes. By default, SFT retrieves $5 \cdot k$ nearest neighbors as candidates in its filtering phase and $\mathcal{K}$NN-distance used by ERkNN is 15NN-distance. The experiments are conducted on a Pentium 4 2.6GHz PC running WinXP. We measure the performance in terms of CPU time, number of node accesses and total query cost which includes both CPU time and I/O overhead by charging each node access 20us [5]. The results are the average of 200 RkNN queries. The query points are randomly picked from the datasets.

4.1 Study of kNN-Distance Estimation

The first set of experiments study the proposed local kNN-distance estimation methods - the PDE method and the kDE method. We estimate the kNN-distance of k=1,2,..., 30 and evaluate the estimation accuracy by the mean square error (MSE).

$$MSE = \frac{\sum_{i=1}^{N}(ednn_k(p_i) - dnn_k(p_i))^2}{N} \quad (12)$$

where N is the number of data points in the dataset.

Figure 4 shows the results on the uniform, Zipf and real life Corel and CA datasets. PDE-id (*or* kDE-id) indicates PDE (*or* kDE) method using the *intrinsic dimensionality*. PDE-ed (*or* kDE-ed) is the PDE (*or* kDE) method using the *embedded dimensionality*. GEM is a global estimation method that calculates the average kNN-distance using the method proposed in [3].

Dataset	Uniform	Zipf	Corel	CA
Embedded	8	8	32	2
Intrinsic	7.2	5.74	6.48	1.96

Table 2: Dimensionality of datasets.

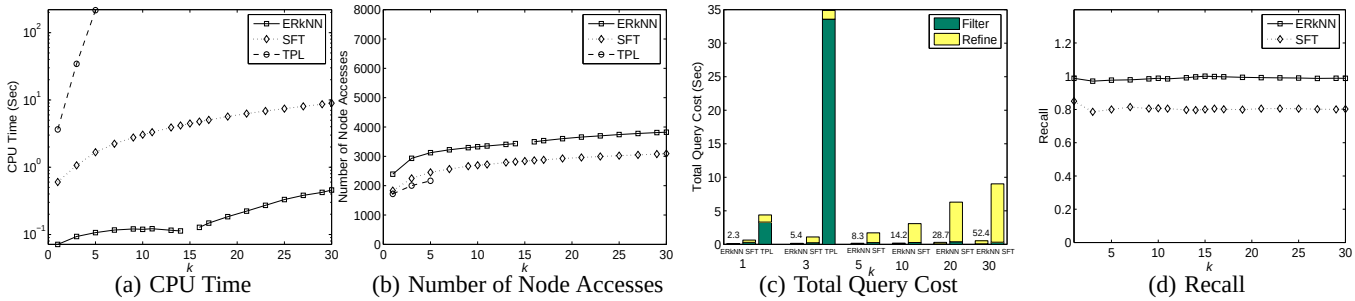
We observe that the PDE and the kDE methods have similar accuracies on all the datasets. Estimations using the intrinsic dimensionality are better than the estimations using the embedded dimensionality. The superiority of PDE-id and kDE-id over PDE-ed and kDE-ed is very clear on the Zipf dataset and the Corel dataset where the intrinsic dimensionality is much lower than the embedded dimensionality (see Table 2). As for the uniform and the CA datasets, their intrinsic dimensionality and embedded dimensionality are similar, so there is not much difference between the estimations using the intrinsic dimensionality and the embedded dimensionality.

The local estimation methods are more accurate than the global estimation method (see Figure 4). The MSE of GEM on Corel dataset is too large to be plotted. On average, the local estimations are 37 times better than GEM on the uniform dataset. The local estimations on the Corel and the Zipf datasets outperform GEM significantly than on uniform and CA datasets.

The study demonstrates that local estimations outperform the global approach significantly and yield more accurate approximation of each point's kNN-distance on both uniformly distributed datasets and real and skewed datasets. The study also confirms that the intrinsic dimensionality captures the effective data dimensionality and leads to better estimations.

4.2 Study on Recall

Next, we evaluate the recall of the answer set retrieved by ERkNN. We query RkNN $k = 10$ and use the PDE method to estimate local kNN-distance. We first evaluate the *local* adjustment. Figure 5 (a) and (b) present the results on the 100K 8-dimensional Zipf datasets. Figure 5(a) exhibits the average recall when we vary the local ad-



(a) CPU Time (b) Number of Node Accesses (c) Total Query Cost (d) Recall

Figure 6: Effect of k (Corel dataset)

justment factor ξ from 1 to 1.06. As expected, the actual recall is always higher than the lower bound. As ξ increases, the recall approaches 1 and the lower bound becomes tighter. Figure 5(b) shows the influence of the local adjustment on the performance of ERkNN in terms of the total query cost. The real number on top of the bars indicate the number of RkNN candidates retrieved. Both the cost of ERkNN and the number of RkNN candidates increase moderately with the increase of ξ.

We evaluate the effect of global adjustment on various datasets. Figure 5 (c) and (d) show the results of the study on the 8-dimensional Zipf dataset. The global adjustment factor λ is varied from 0 to 0.015. Our study finds that the global adjustment has the similar influence on the recall and performance of ERkNN as the local adjustment on the uniform data but works more effectively on the skewed Zipf dataset. On the Zipf dataset, when ERkNN with local adjustment reaches 100% recall, only 15 RkNN candidates are retrieved. While ERkNN with global adjustment needs to retrieve 22 candidates. The reason is the local adjustment is more adaptive to the data density distribution. The experiment shows that the recall of ERkNN can be adjusted effectively and the adjustments affect the performance of ERkNN moderately.

4.3 Study on Real Datasets

We now compare the performance of ERkNN with SFT and TPL on real dataset with varying values of k. Figure 6 presents the results on the Corel datasets when we vary k from 1 to 30. Note that the lines of ERkNN have a break at k=15 because when $k = \mathcal{K}$ the RkNN queries are answered using the point enclosure query directly. Contrary to the experiment results in [16], SFT is more efficient than TPL in our experiments. The reasons are two-fold. First, SFT retrieves only $5 \cdot k$ points as candidates in our experiments, while SFT retrieves $10 \cdot d \cdot k$ points as candidates in the experiments in [16]. Second, we use an optimized SFT with batch execution of the boolean range queries [13]. The batch execution of boolean range queries reduces I/O cost and speeds up the query performance considerably [13].

This study shows that ERkNN outperforms both TPL and SFT significantly. The speed-up factor on the Corel dataset in terms of total response time is 50.5 when k is 1 and 2024.45 when k is 3. The average speed-up of ERkNN over SFT is more than 20. TPL is expensive when k is large mainly because the *k-trim* algorithm used by TPL to prune an R-tree node requires to do $\binom{n_c}{k}$ times *clippings* [16], where n_c is the number of RkNN candidates. ERkNN is more efficient than SFT because its low CPU cost for candidates retrieval and refinement. During the filtering phase, ERkNN performs the point enclosure query, while SFT performs the kNN query. kNN query is more expensive due to the additional CPU cost to sort and insert the kNN candidates. It is considerable especially when k is large. The refinement procedure of ERkNN

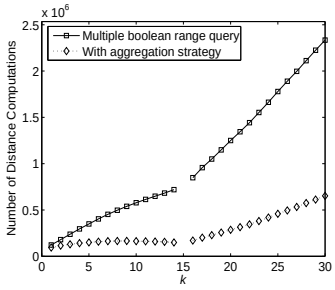


Figure 7: Number of distance computation on Corel dataset

is also more efficient because ERkNN retrieves much fewer candidates than SFT and the *aggregation strategy* employed by ERkNN prunes a large number of distance computations, thus reducing the CPU cost greatly. Experiment results on the Corel dataset show that the *aggregation strategy* prunes around 75% distance computations on average (see Figure 7).

We observe that ERkNN incurs more node accesses than SFT. This is because ERkNN accesses more nodes during the filtering phase (ERkNN accesses 1863.95 nodes for the Corel dataset while SFT accesses 1319.65 nodes on average). The reason is that ERkNN may access more tree nodes to retrieve some potential RkNNs which are far from the query point according to the estimated kNN-distance. Further, the lowest recall of ERkNN is 97.12% on the Corel dataset. The lowest recall of SFT is 79.6%.

The study also shows that for RkNN query, I/O cost is no longer the dominant cost for both SFT and ERkNN because both methods execute a set of range queries simultaneously and traverse the index tree only once in the refinement procedure. CPU cost is more expensive because there are multiple candidates to be verified. On Corel data, the average I/O overhead of ERkNN and SFT is 0.069 sec (3342.26 node accesses) and 0.055 sec (2763.46 nodes accesses) respectively, while the CPU cost is 0.19 sec and 4.68 sec.

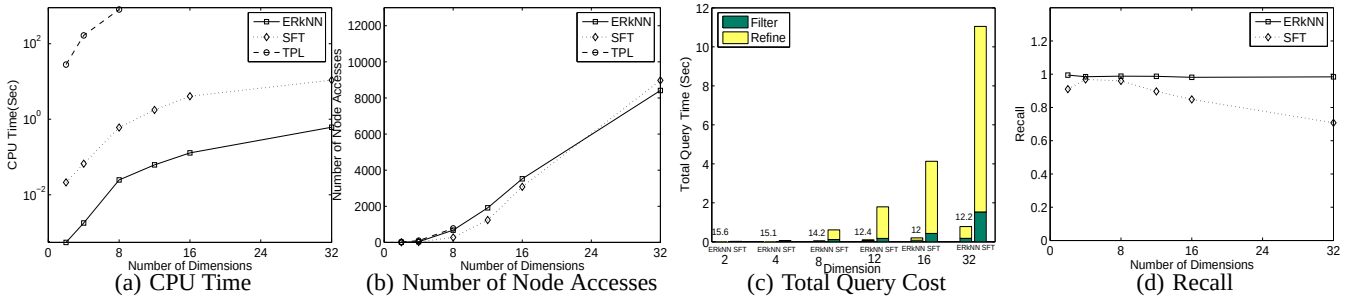
4.4 Study on Synthetic Datasets

In this section, we study ERkNN, SFT and TPL on synthetic datasets of various sizes and dimensions.

4.4.1 Effect of Dimensionality

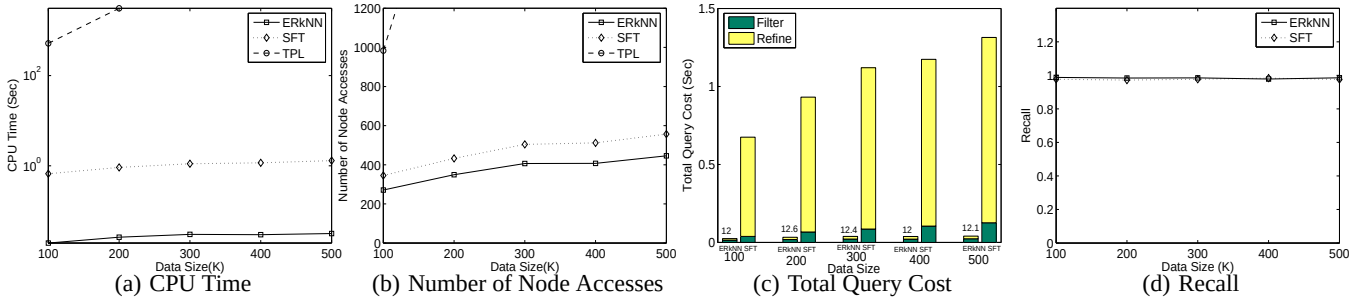
We first evaluate the effect of data dimensionality on the RkNN query by varying the number of dimensions from 2 to 32 (k = 10). Figure 8 shows the results on the Zipf datasets.

ERkNN outperforms SFT by an average factor of 28.7 on Zipf datasets. The recall of ERkNN remains higher than that of SFT. For the Zipf datasets, the recall of ERkNN remains steady at around 99%, while the recall of SFT decreases from 91.03% to 70.79% when the data dimensionality increases from 2 to 32. The study demonstrates that ERkNN is more scalable to RkNN queries in high-dimensional spaces compared to TPL and SFT.



(a) CPU Time (b) Number of Node Accesses (c) Total Query Cost (d) Recall

Figure 8: Effect of Data Dimensionality (Zipf, 100K)



(a) CPU Time (b) Number of Node Accesses (c) Total Query Cost (d) Recall

Figure 9: Effect of Data Size (Uniform, Dim=8)

4.4.2 Effect of Data Size

We examine the RkNN query performance on datasets of varying sizes. We query RkNN $k = 10$ on the uniform datasets and vary the dataset size from 100,000 to 500,000 objects. Figure 9 shows the results. We observe that ERkNN outperforms SFT and TPL significantly. In terms of total query cost, the average speed-up of ERkNN over SFT is 29.6 on the uniform datasets. The average speed-up of ERkNN over TPL is over 10,000. Figure 9(d) compares the recalls of the RkNN answer sets retrieved by ERkNN and SFT. As in the other studies, ERkNN has a higher than SFT.

5. CONCLUSION

RkNN queries have important applications in many database systems. However, existing methods are expensive and not scalable to RkNN queries in high dimensional space or of large values of k. In this paper, we propose an efficient *estimation-based* approach - ERkNN. It employs an estimation-based filter which requires much lower computation cost. In order to estimate the kNN-distance accurately, we propose the *local kNN-distance estimation* methods. Extensive experiments demonstrate that ERkNN is efficient, scalable and outperforms pervious methods significantly.

6. ACKNOWLEDGMENTS

This work was supported by A*STAR-NUS grant R-252-000-172-593. We thank authors of [16] and [18] for providing the codes.

7. REFERENCES

[1] http://kdd.ics.uci.edu/.

[2] http://s2k-ftp.cs.berkeley.edu:8000/sequoia/benchmark/.

[3] C. Böhm. A cost model for query processing in high dimensional data spaces. *TODS*, 25(2):129–178, 2000.

[4] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[5] L. Chung, J. Gray, B. Worthington, and R. Horst. *Windows 2000 Disk IO Performance*. http://research.microsoft.com/.

[6] K. Fischer. *Smallest enclosing ball of balls. Diploma thesis, Institute of Theoretical Computer Science*. ETH Zurish, 2001.

[7] K. Fukunaga. *Introduction to Statistical Pattern Recognition (2nd edition)*. Academic Press, 1990.

[8] N. Katayama and S. Satoh. Distinctiveness-sensitive nearest-neighbor search for efficient similarity retrieval of multimedia information. In *ICDE*, pages 493–502, 2001.

[9] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *STDM*, pages 119–134, 1999.

[10] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *ACM SIGMOD*, pages 201–212, 2000.

[11] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse nearest neighbor aggregates over data streams. In *VLDB*, 2002.

[12] K.-I. Lin, M. Nolen, and C. Yang. Applying bulk insertion techniques for dynamic reverse nearest neighbor problems. In *IDEAS*, pages 290–297, 2003.

[13] A. Singh, H. Ferhatosmanoglu, and A. Ş. Tosun. High dimensional reverse nearest neighbor queries. In *CIKM*, pages 91–98, 2003.

[14] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[15] I. Stanoi, M. Riedewald, D. Agrawal, and A. El Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*, pages 99–108, 2001.

[16] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, pages 744–755, 2004.

[17] N. Wyse, R. Dubes, and A.K. Jain. A critical evaluation of intrinsic dimensionality algorithms. *Pattern Recognition in Practice*, pages 415–425, 1980.

[18] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, pages 485–492, 2001.