Processing Multiple *k* Nearest Neighbor Queries

I-Fang Su, Yu-Chi Chung, Pei-Chi Liu, and Chiang Lee

***Abstract*—*k* Nearest Neighbor (*k*NN) query has received considerable attention from the database and information retrieval communities. The applications of *k*NN query are widely, not only in spatio-temporal database but also in many areas. The past studies of *k*NN query processing did not consider the case that the server may receive multiple *k*NN queries at a time. Their algorithms process queries independently. Thus, server will busy with continuously re-accessing database to obtain the data that already acquired. It results in wasting I/O cost and degrading the performance of whole system. In this paper, we focus on this problem and propose an algorithm that namely Multiple *k*NN Search. The main idea of this problem is "information sharing" strategy that server re-uses the query results of previously executed queries for efficiently processing subsequent queries. We conduct a comprehensive set of experiments to analyze the performance of Multiple *k*NN Search and compare it with Best-First Search (BFS) algorithm. Empirical studies indicate that the performance of Multiple *k*NN Search outperforms BFS, achieves lower I/O cost and less running time.**

***Index Terms*—*k* Nearest Neighbor, Multiple *k*NN query, spatio-temporal database, information sharing strategy, query processing.**

I. Introduction

The issue of *k* nearest neighbor (*k*NN) queries has been widely researched in the past few years [1]-[7]. Given a static data set *D* with a dimensionality of *d*, when a user sends a *k*NN query *q* to a server, the server searches for the *k* nearest neighbors of *q* within *D* and sends them back to the user. However, most existing studies focus on the processing of single *k*NN queries. In reality, a number of similar *k*NN queries may be issued, and many users may issue similar queries during a period of time. For example, during a vacation season, the database management systems (DBMS) of tourism websites receive a large number of interdependent *k*NN queries. An example is a query about hotel rates, in which q_1 is a query for five hotels with ratings near 2 stars and accommodation rates near $70, and q_2 is a query for three hotels with ratings near 2.5 stars and accommodation rates near $78. In this example, although q_1 and q_2 are issued from different users, they are correlated (that is to say, these two queries are quite similar to each other). There are also many examples of the correlations that exist among queries in data mining applications.

Manuscript received May 6, 2013; revised July 26, 2013. This work is supported by National Science Council of Taiwan (R.O.C) under Grants NSC 101-2221-E-268-006, NSC 101-2221-E-309-009 and NSC 100-2221-E-006-249-MY3.

I-Fang Su is with MIS of Fortune Institute Technology, Taiwan (e-mail: emily@csie.ncku.edu.tw).

Yu-Chi Chung and Chiang Lee are with CSIE of Chang Jung Christian University, Taiwan (e-mail: justim@mail.cjcu.edu.tw, leec@mail.ncku.edu.tw).

Pei-Chi Liu is with Chunghwa Telecom Laboratories, Taiwan (e-mail: ayumi416@imus.csie.ncku.edu.tw).

Most of the existing algorithms process each *k*NN query independently and do not consider the correlations or intersections among *k*NN queries so that many similar queries are processed repeatedly, causing a significant waste of computing resources. In this paper, we propose a *k*NN query processing algorithm for "multiple" *k*NN queries. The proposed algorithm identifies the correlations among multiple *k*NN queries and utilizes the correlations to speed-up the processing of multiple *k*NN queries.

Few previous studies [8], [9], [12] have considered the problem of multiple *k*NN queries. Most of them have used indexing techniques (e.g., *R*-tree [10], *R**-tree) and a shared execution mechanism to enhance the processing performance of multiple *k*NN queries. Fig. 1 explains how the existing methods work. For ease of presentation, it is assumed that the search regions of q_1 and q_2 (i.e., the circles in Fig. 1(a)) are already known. In this example, the data points are indexed by an R-tree, as shown in Fig. 1 (b). Suppose that q_1 is processed first. The query processing algorithm traverses the R-tree and accesses the nodes that overlap the search region of q_1. When retrieving the data point *p*, the shared execution mechanism not only determines whether *p* is inside the search region of q_1 but also tests if *p* lies in the search region of q_2. The benefit of the shared execution mechanism is that after q_1 is accomplished, the answer of q_2 is also partially determined. The I/O cost can be greatly reduced when processing q_2, because many of the data pages that are required when processing q_2 are already inside the memory.

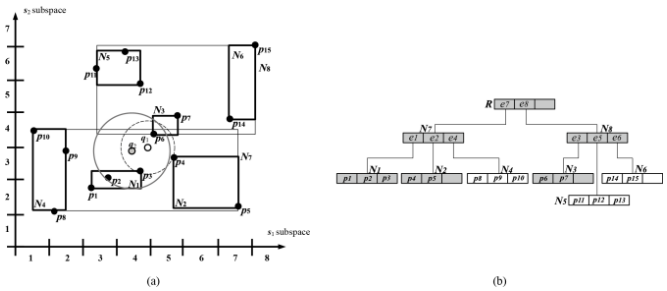

Fig. 1. Example of the correlation or intersection between queries.

One limitation of the existing methods is that the index tree will be repeatedly accessed, which incurs excessive computation costs. In the previous example, the *R*-tree has to be accessed twice, even if q_1 and q_2 are related to each other. While visiting the R-tree, the distances between the query point and the internal nodes must be calculated to determine the order in which the internal nodes will be visited. This step requires a considerable amount of CPU computation. In Fig. 1 (b), the nodes that are accessed by q_1 and q_2 are shown shaded. There are a total of 11 distance computations (i.e., q_1 accesses nodes *R*, N_7, N_8, N_1, N_2, and N_3; q_2 accesses *R*, N_7, N_8, N_1, and N_3).

The algorithm proposed in this paper, called the **CORrelated kNN query processing** (CORK) algorithm,

avoids the above limitation. CORK contains an innovative data structure, known as RGrid. In processing *k*NN queries, R-tree is perhaps the most well-known and popularly used indexing structure in the literature. Hence, RGrid used in this paper is an index structure constructed on an *R*-tree, which is employed to reduce the computation cost while repeatedly visiting the R-tree. Given a search region of q, an RGrid enables direct access to the leaf nodes of the R-tree and a quick extraction of the data inside the search region of q. Thus, visiting the internal nodes of the R-tree can be avoided. Bitmaps are used to implement the RGrid so that answering *k*NN queries becomes a series of bitwise operations (e.g., bitwise-AND and bitwise-OR) on the bitmaps. Since bitwise operations are fast and can often be accelerated by low-level hardware instructions, the CPU computation cost of using RGrid will be much less than that of visiting an R-tree. In Fig. 1(b), which shows an RGrid, the number of distance computations can be reduced from 11 to 8 (i.e., q_1 accesses nodes R, N_7, N_8, N_1, N_2, and N_3; q_2 only accesses N_1, and N_3). The more related queries that are executed, the greater the amount of distance computation cost can be saved by using the RGrid.

In this study, we compare the efficiency of the proposed method with that of the existing *k*NN algorithms (e.g., BFS [1]). The experimental results demonstrate that the proposed method is superior to those *k*NN algorithms in both computation cost and disk access cost.

The remainder of the paper is organized as follows: Section II gives a brief review of related work. Section III presents the problem definition and several terms that will be used in the CORK algorithm. The details of the CORK algorithm are explained in Section IV. Section V presents the experimental results, followed by the conclusion in Section VI.

II. Related Works

Zhuang *et al*. [12] proposed the multi-query optimization technique for distributed similarity query processing (MDSQ) to speed up the multi-query processing performance in distributed environments. When receiving a number of simultaneously submitted queries, MDSQ identifies the correlation among the queries and then groups related queries together. MDSQ then invokes a shared execution mechanism to process the grouped queries. Fig. 2 illustrates the functionality of the shared execution mechanism with two grouped queries, q_1 and q_2, and their query spheres. The advantage of the shared execution mechanism is that if each query is evaluated independently, each query performs a disk scan to identify the data points within its query sphere. Thus, the data points residing in the overlapping area of q_1 and q_2 (i.e., the shaded area in Fig. 2) will be accessed twice, which increases the disk access cost. With shared execution, there is only one scan over the disk. Thus, excessive disk access can be avoided. Scalable incremental hash-based algorithm (SINA), proposed by Mokbel *et al*. [11] also utilizes the similar shared execution paradigm to reduce the cost of multiple range query processing. However, SINA deals with continuous range queries, while MDSQ focuses on snapshot range query processing.

Braünmuller *et al.* [8] developed a multiple queries processing algorithm (MQP) to efficiently process sets of simultaneously issued *k*NN queries. MQP uses an incremental processing scheme to speed-up the processing of multiple *k*NN queries. Suppose MQP receives two *k*NN queries, q_1 and q_2. When processing q_1, MQP not only determines whether a data point p is an answer for q_1 but also checks if p is relevant for q_2. Consequently, after q_1 is accomplished, the answer for q_2 will be partially determined. When processing q_2, the data pages that are read when processing q_1 do not need to be loaded again, therefore reducing the disk access cost. If the incremental processing scheme is not used, the data pages relevant to both q_1 and q_2 will have to be loaded twice. Clearly, this will cause a higher number of disk I/Os.

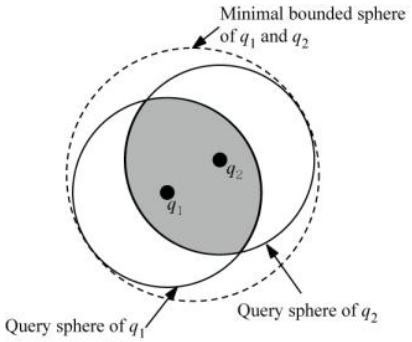

Fig. 2. An example of the shared execution mechanism.

Another shared execution algorithm (SEA-CNN) proposed by Xiong *et al.* [9] is able to process continuous spatio-temporal *k*NN queries. The performance of SEA-CNN is achieved by employing two important modules: (1) the incremental evaluation module, and (2) the shared execution module. The incremental evaluation module can efficiently determine whether a continuous *k*NN query q should be re-evaluated or not. If the answer for q is not affected by the motion of the data points (or other queries), then the re-evaluation of q can be ignored, which saves the computation cost. The idea of a shared execution module is similar to the one mentioned earlier. The shared execution module can group concurrently running queries together and process them efficiently.

III. Prelimanary

This section describes the environment and assumptions in this study and provides a formal definition of the multiple *k*NN search problem. Given a d-dimensional space $S=\{s_1, s_2,\ldots,s_d\}$, $D=\{p_1, p_2,\ldots,p_n\}$ is a data set in S, in which every $p_i \in S$ is a d-dimensional data point in S. This study used $p_i[j]$ to represent the jth-dimensional attribute value of data item p_i. Let $Q=\{q_1, q_2,\ldots,q_m\}$ be a *k*NN query set, in which q_i is the ith *k*NN query, and q_i. k is the k value requested by q_i. The queries within Q are ordered according to their arrival time. Note that the queries in Q are not required to be submitted to the server at the same time. The attribute values of each data point and each query point are normalized to the range [0..1]. This study used two-dimensional data (i.e, $d = 2$); however, the proposed approach could be generalized to a multidimensional space in a straightforward manner.

This paper aimed at solving the following problem:

Given a set of *k*NN queries Q submitted to the server, how

can Q be answered with the minimal computation cost and the minimal disk access cost?

IV. Correlated *k*NN Query Processing Algorithm

While receiving a query q, we first determine whether q can be processed by CORK. CORK can only be used to process q when a correlated query (***Corr***(***q***)) provides sufficient information to speed up the processing of q. Sufficient information refers to the number of data points in the union of *k*NN results for all of the correlated queries of q (denoted as $|k\text{NN}(\text{Corr}(q))|$) being greater than or equal to $q.k$ (i.e., $|k\text{NN}(\text{Corr}(q))| \geq q.k$). If so, CORK retrieves the k-th farthest data point p from q in kNN(Corr(q)). Then, $dist(p, q)$ is the radius of $SR(q)$, where $dist(p, q)$ is the Euclidean distance between p and q. $SR(q)$ must encompass kNN(q). If CORK cannot accelerate the processing of q, a conventional *k*NN processing algorithm is employed to process q. In this study, the best first search (BFS) algorithm [1] was used to obtain kNN(q).

Like many existing *k*NN query processing algorithms [1, 2], this study employed R-trees to index the data points. To quickly obtain the leaf nodes of the R-tree that intersect with $SR(q)$, we develop an innovative approach, called RGrid, to reduce the computational penalty by eliminating the need to visit the internal nodes of the R-tree. RGrids is capable of efficiently identifying the MBRs that intersect with the search region. In an RGrid, a d-dimensional data space S is divided into d 1-dimensional subspaces. This study defined s_i to be the ith subspace of S. In Fig. 3 (a) and 3(b), a 2-dimensional data space is split into two 1-dimensional spaces s_1 and s_2. Each subspace is further divided into δ individual partitions (δ is a system parameter). For example, in Fig. 3(a), subspace s_1 is divided into eight partitions (i.e., δ=8). Each partition is associated with an MBR ID list, which stores the IDs of the MBRs that overlap the partition. For example, in Fig. 3(a), N_1 intersects with partitions 3 and 4 in s_1. Hence, the ID N_1 is inserted into the MBR ID lists associated with partitions 3 and 4 in Fig. 3(a). To identify the MBRs that intersect with a search region SR(q), it is necessary to find out the partitions of each subspace that intersect with SR(q). The search result can be obtained from the associated MBR ID lists.

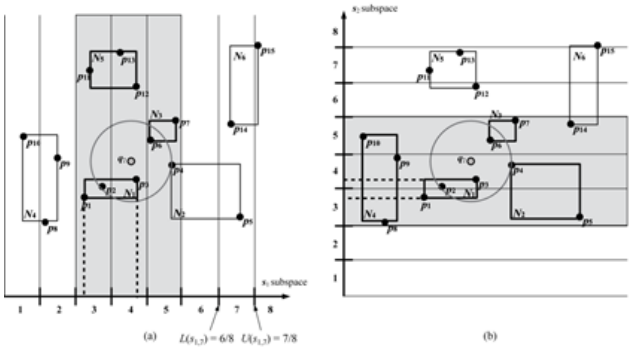

Fig. 3. RGrid example.

Suppose N_i is a leaf node of the R-tree; $N_i.l$ and $N_i.u$ are the lower-left and upper-right corners of N_i. The j-dimensional attribute values of $N_i.l$ and $N_i.u$ are represented as $N_i.l[j]$ and $N_i.u[j]$. The d-dimensional space $S=\{s_1, s_2, \ldots, s_d\}$ is divided into d subspaces. Each subspace s_i (i = 1, 2, …, d), is further divided it into δ partitions (δ is a system parameter), with $s_{i,j}$ representing the jth partition of s_i. The attribute values of each dimension are normalize to the range [0, 1]. As a result, the width of each partition is $1/\delta$. Let $L(s_{i,j})$ and $U(s_{i,j})$ be the lower bound and upper bound values of $s_{i,j}$. It can easily be found that $L(s_{i,j}) = (j-1)/\delta$ and $U(s_{i,j}) = j/\delta$.

Each subspace is associated with an RGrid. In other words, if S is a d-dimensional space, then d RGrids will be established, and RGrid(s_i) will represent the RGrid of s_i. RGrid(s_i) is implemented with δ vectors. RGrid (s_i)[j] is used to denote the jth vector of RGrid(s_i). The length of RGrid(s_i)[j] is equal to the number of leaf nodes of the indexed R-tree. RGrid(s_i)[j][k] is the kth item of RGrid(s_i)[j]. Given a leaf node N_k and the j-th partition of s_i (i.e., $s_{i,j}$), the relationship between N_k and $s_{i,j}$ is stored in RGrid(s_i)[j][k]. MBR($SR(q)$, s_i) is also defined to be a set of MBRs that intersect with $SR(q)$ in s_i. Let MBR($SR(q)$) be a set of the covering MBRs of $SR(q)$ in the d-dimensional space. As described before, it can be derived that MBR($SR(q)$) = MBR($SR(q)$, s_1)∩ MBR($SR(q)$, s_2) ∩ ... ∩MBR($SR(q)$, s_d). For example, in Fig. 3, MBR($SR(q_2)$, s_1) = {N_1, N_2, N_3, N_5} and MBR($SR(q_2)$, s_2) ={N_1, N_2, N_3, N_4, N_6}. Thus, MBR($SR(q_2)$) = MBR($SR(q_2)$, s_1)∩ MBR($SR(q_2)$, s_2) = { N_1, N_2, N_3}. As illustrated in Fig. 3, a leaf node N_k has three possible relationships with $s_{i,j}$. (1) N_k appears before $s_{i,j}$, (2) N_k appears after $s_{i,j}$, (3) N_k intersects $s_{i,j}$. We use 10, 01 and 00 to represent this three relationships, respectively. Returning to the example in Fig. 3, the context of RGrid(s_1) is presented in Fig. 3(a) is shown in Table I. In this example, the R-tree contains six leaf nodes; therefore, each RGrid(s_i) [j] has six items.

TABLE I: RGrid(s_1) of Fig. 3(a)

	N_1	N_2	N_3	N_4	N_5	N_6
RGrid(s_1)[1]	01	01	01	00	01	01
RGrid(s_1)[2]	01	01	01	00	10	01
RGrid(s_1)[3]	00	01	01	10	00	01
RGrid(s_1)[4]	00	01	01	10	00	01
RGrid(s_1)[5]	10	00	00	10	10	01
RGrid(s_1)[6]	10	00	10	10	10	01
RGrid(s_1)[7]	10	00	10	10	10	00
RGrid(s_1)[8]	10	10	10	10	10	00

Next, we explain how to use RGrids to quickly identify the intersecting MBRs of $SR(q)$. Given a query region $SR(q)$, the projection of $SR(q)$ on the ith subspace s_i is the interval [$q[i]$-r, $q[i]$+r], in which $q[i]$ and r are respectively the ith dimension attribute of q and the radius of $SR(q)$. Let $s_{i,l}$ and $s_{i,u}$ correspond to the covering partitions of $q[i]$-r and $q[i]$+r. l and u can be easily derived using the following formula:

$$l = \lceil \text{MAX}(q[i] - r, 0) \times \delta \rceil$$

$$u = \lceil \text{MIN}(q[i] + r, 1) \times \delta \rceil$$

Taking Fig. 3(a) as an example, suppose r = 0.125, $q_2[1]$ = 0.4375, and $\delta = 8$. The projection of $SR(q_2)$ on first dimension is the interval [$q_2[1]$ - 0.125, $q_2[1]$ + 0.125] = [0.3125, 0.5625]. As a result, $l = \lceil \text{MAX}(0.3125, 0) \times 8 \rceil = \lceil 2.5 \rceil = 3$. Similarly, it can be derived that $u = \lceil \text{MIN}(0.5625, 1) \times 8 \rceil = 5$. Therefore, the covering partitions of $q_2[1]$-r and $q_2[1]$-r are $s_{1,3}$ and $s_{1,5}$, respectively.

Given a R-tree leaf node N_k, the following observations can be made:

Observation 1: If N_k does not intersect $SR(q)$ on s_i, then N_k

satisfies either Condition 1 or Condition 2, as described below.

Condition 1: N_k appears before $s_{i,l}$. In Fig. 3(a), N_4 does not intersect *SR(q2)*, because N_4 appears before $s_{1,l}$=$s_{1,3}$.

Condition 2: N_k appears after $s_{i,u}$. In Fig. 3(a), N_6 does not intersect *SR(q2)*, because N_6 appears after $s_{1,u}$=$s_{1,5}$.

Observation 1 allows a quick identification of the MBRs that intersect *SR(q)* on a given subspace s_i. Let RGrid(s_i)[l] and RGrid(s_i)[u] be the corresponding vectors of $s_{i,l}$ and $s_{i,u}$, respectively. Each element of RGrid(s_i)[l] and RGrid(s_i)[u] are scanned in a parallel fashion. If RGrid(s_i)[l][k]=10 (i.e., Condition 1) or RGrid(s_i)[f][k]=01 (i.e., Condition 2), then N_k can be safely pruned because it does not intersect *SR(q)* on subspace s_i. By the above conditions, we can derive a bit vector, *Result_Vector*, which contains b bits (where b is the number of leaf nodes in an R-tree). The kth bit of *Result_Vector* is set to 0 if N_k satisfies either Condition 1 or Condition 2 (i.e., N_k does not intersect *SR*(q) on s_i); otherwise it is set to 1 (i.e., N_k intersects *SR*(q) on s_i). For example in Fig. 3(a), the corresponding vectors of $s_{1,3}$, and $s_{1,5}$ on RGrid(s_i) are RGrid(s_1) [3] and RGrid(s_1) [5], respectively. It is then possible to derive *Result_Vector*=[1, 1, 1, 0, 1, 0], meaning that MBRs N_1, N_2, N_3, and N_5 intersect *SR*(q_2) on s_1.

We then use a bit vector, *MBR_Vector*, which is obtained by intersecting all of the d returned bit vectors. If the kth bit of *MBR_Vector* is 1, it indicates that N_k intersects with *SR*(q) in S. For example, in Fig. 3(a) and Fig. 3(b), the *Result_Vectors* of s_1 and s_2 are [1,1,1,0,1,0] and [1,1,1,1,0,1]. This obtains *MBR_Vector*=[1,1,1,0,1,0] ⋀ [1,1,1,1,0,1]=[1,1,1,0,0,0] (where ⋀ is a bitwise-AND operation). Thus, MBRs N_1, N_2, and N_3 intersect with *SR*(q_2) in the 2-dimensional space. Each bit of the MBR_Vector is sequentially scanned. If the kth bit in *MBR_Vector* is 1, then MBR N_k is an intersecting MBR of *SR*(q). MPT[k] is then visited and the pointer stored in it is followed to obtain N_k. For example, the *MBR_Vector* is [1,1,1,0,0,0]. Next, MPT [1], MPT [2] and MPT [3] are visited to retrieve the instances of N_1, N_2 and N_3.

Following is an explanation of how to identify kNN(q) from the set of intersecting MBRs of *SR*(q). In the CORK algorithm, we maintain two queues, namely, the *candidate queue* and the *result queue*. In the beginning, all the intersecting MBRs of *SR*(q) are placed within the candidate queue, in which each MBR N_k is sorted in increasing order of its *mindist* (i.e., the minimum distance [1]) from q. In each iteration, Algorithm accesses the MBR at the top of the candidate queue (denoted by N_{top}). Then, Algorithm extracts every data point p contained in N_{top} and checks if p is within *SR*(q). If p is inside *SR*(q), then p is added into the result queue, where each entry is sorted in ascending order of its distance from q. At the end of each iteration, Algorithm will shrink the search region if it finds that the distance of q from the kth entry in the result queue is smaller than the current search radius r. The search stops when the minimum distance of the top MBR in the candidate queue is greater than the current search radius r. This means that all data points contained in the unseen MBRs have a distance larger than r; thus, they cannot belong to kNN(q). Continuing the example of Fig. 3, assume that r = dist(p_4, q_2) and the returned MBR_Vector is [1,1,1,0,0,0]. Thus, {N_1, N_3, N_2} are inserted into the candidate queue. Row 1 of Table II exhibits the context of the candidate queue, the result queue, and r. In Step 1, CORK accesses the top entry (i.e., N_{top}) of the candidate queue. In this case, N_{top} = N_1. Since the data points {p_3, p_2} of N_1 lie in the search region *SR*(q_2), they are put into the result queue. The context of each data structure after the execution of Step 1 is listed in Row 2 of Table II. In Step 2, CORK extracts N_3 from the candidate queue and checks the data points within N_3 to see if they fall in *SR*(q_2). Since dist(p_6, q_2) is less than the current search radius, p_6 is added into the result queue. On the other hand, dist(p_7, q_2) > r, meaning that p_7 does not belong to kNN(q_2). Thus, p_7 can be safely pruned. At the end of Step 2, CORK alters the current search radius r to dist(p_2, q), because the distance of q_2 from the third element of the result queue (i.e., p_2) is found to be less than r. The results obtained after completing Step 3 are shown in Row 3 of Table II. Now, as the *mindist*() of the first entry of the candidate queue (i.e., N_2) is greater than r, the data points included in N_2 (i.e., {p_4, p_5}) fall outside of *SR*(q_2). Consequently, they cannot become kNN(q_2), and therefore the algorithm terminates.

TABLE II: EXECUTION PROCEDURE OF CORK

step	result queue	candidate queue	r
Initial	ϕ	{N_1, N_3, N_2}	dist(p_4, q_2)
1	{p_3, p_2}	{N_3, N_2}	dist(p_4, q_2)
2	{p_3, p_6, p_2}	{N_2}	dist(p_2, q_2)

V. EXPERIMENTS

A. Performance Settings

In this section, we conducted several experiments to evaluate the performance of CORK and BFS. In each experiment, we compare our algorithm with BFS in terms of the average number of disk I/O and the CPU time. In BFS, each kNN query q is processed independently while in our method is employed to accelerate the processing of q if its correlated queries are exist.

A synthetically generated (SG) dataset is used in the following experiments. The data point distribution of SG is uniform while that of NE is highly skewed. The dimension, d, for each data point is ranged from 2 to 6. The random cluster (RC) query set is used in the simulation. In the RC query set, we first randomly select 10 cluster centers in the d-dimensional space. We use *cluster*$_i$ to denote the i-th (i=1, 2, ..., 10) cluster center. For each *cluster*$_i$, we randomly generate k_i queries so that each query q in *cluster*$_i$ satisfies the following condition: the distance from q to *cluster*$_i$ is less than ε. This condition ensures that each q belonging to *cluster*$_i$ is not too far from *cluster*$_i$. In this paper, we set ε=0.15. Clearly, if two queries, q_i and q_j, are inside the same cluster, then they are correlated queries. We use RC query set to show that the idea to use the information provided by the correlated queries can facilitate efficient kNN query processing. We also set a k value to each query. The value of k is chosen uniformly at random from [1, 100].

The simulations are performed in the following way. We first generate |Q| kNN queries. The algorithm (i.e., BFS or CORK) then evaluates each query one by one. For each query, our simulator records the execution time and the number of nodes accessed of the algorithm. After processing |Q| queries, the simulator produces the total CPU time and the total I/O

cost.

The parameters and default values are summarized in Table III.

TABLE III: THE SETTING OF THE PARAMETERS IN CORK

Parameter	Default Value	Range
Number of Objects (|*D*|)	100K	20K, 40K,60K,80K,100K
Number of Queries (|*Q*|)	8K	1K,4K,8K,12K,14K
Data Grid Granularity δ	40	10,20,30,40,50,60,70,80,90,100

1) Effect of number of queries

In this section, we study effect of number of queries. Fig. 4(a) and Fig. 4(b) respectively plot CPU time and I/O cost against the number of queries (|Q|) from 0.1K up to 10K for Random Cluster query sets while the data cardinality (|D|) is fixed at 100K. CORK performs better with the increasing query cardinality. As the number of queries increases, CORK has a higher opportunity to find the correlated queries of a given query *q*. Therefore, the technique proposed in the paper can be adapted to reduce CPU time and I/O cost for the processing of *q*.

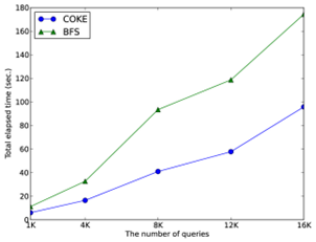

Fig. 4(a). CPU time.

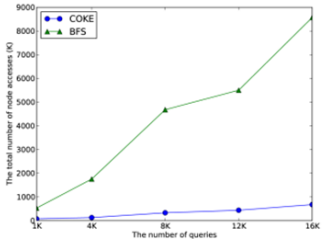

Fig. 4(b). I/O cost.

2) Effect of number of data points

In this experiment, we study the effect of number of data points. Fig. 5 shows the performances of CORK and BFS with the data cardinalities (i.e., |*D*|) varied from 20K to 100K and |*Q*| and *d* respectively fixed at 6K and 2. Again, CORK shows its superiority over BFS. With the increase of the data size, the execution time and the I/O cost of BFS grows faster than CORK. When the number of data points increases, BFS accesses more data pages and performs more distance computations in order to find the *k*NN result of a query. On the other hand, in CORK, the use of RGrid can successfully reduce the number of distance computations and page accesses, making CORK performs better than BFS. The experiment also demonstrates that CORK is more scalable than BFS for large datasets.

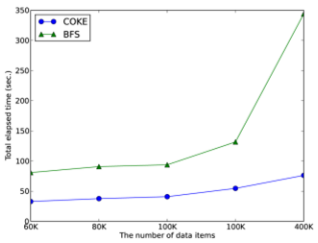

Fig. 5(a). CPU time.

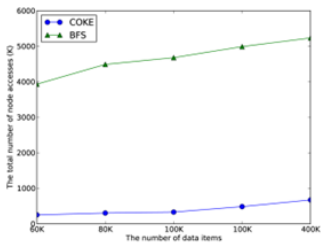

Fig. 5(b). I/O cost.

VI. CONCLUSION

In this paper, we introduced a technique that can quickly identify the related queries of a given query *q*. After finding the related queries of *q*, our algorithm uses the information to infer a search region that covers the *k*NN results of *q*. We employ an R-Grid to reduce the distance computations when traversing an R-tree. The performance results show that the techniques proposed in the paper can speed up the performance and reduce the disk I/O cost of *k*NN searching. Currently, our algorithm can only deal with *k*NN queries. In the future, we plan to extend our work to permit various spatial queries (e.g., reverse *k*NN, aggregate NN and range queries). We plan to explore this subject further in the future.

REFERENCES

[1] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *TODS*, June 1999.
[2] N. Roussopoulos, S. Kelly, and F. Vncent, "Nearest neighbor queries," *ACM SIGMOD*, May 1995..
[3] M. Kolahdouzan and C. Shahabi, "Voronoi-based *k* nearest neighbor search for spatial network databases," *VLDB*, August 2004, pp. 840–851.
[4] K. Mouratidis and D. Papadias, "Continuous nearest neighbor queries over sliding windows," *TKDE*, June 2007..
[5] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan, "Indexing high-dimensional data for efficient in-memory similarity search," *TKDE*, vol. 17, March 2005.
[6] H. Lu, B. C. Ooi, H. T. Shen, and X. Xue, "Hierarchical indexing structure for efficient similarity search in video retrieval," *TKDE*, vol. 18, pp. 1544–1559, 2006.
[7] C. Bohm, B. C. Ooi, C. Plant, and Y. Yan, "Efficiently processing continuous k-nn queries on data streams," *ICDE*, April 2007.
[8] B. Braünmuller, M. Ester, H.-P. Kriegel, and J. Sander, "Multiple Similarity Queries: A Basic DBMS Operation for Mining in Metric Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 1, PP. 79-95, 2001.
[9] X. Xiong, M. F. Mokbel, and W. G. Aref, "SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases," in *Proc. International Conference on Data Engineering (ICDE)*, April 2005, pp. 643–654.
[10] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *.ACM SIGMOD*, June 1984.
[11] M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases," *ACM SIGMOD*, June 2004, pp. 623–634.
[12] Y. Zhuang, Q. Li, and L. Chen, "Multi-query Optimization for Distributed Similarity Query Processing, in *Proc. 28th International Conference on Distributed Computing Systems (ICDCS)*, 2008.

I-Fang Su received her Ph.D. degree in the Dept. of Computer Science and Information Engineering at the National Cheng-Kung University, Taiwan, in 2010. Currently, she is an assistant professor of the Dept. of Information Management, Fotech, Taiwan. Her research interests include mobile data management, sensor networks, skyline query processing, spatio-temporal databases, and web information retrieval.

Yu-Chi Chung received his Ph.D. degree in the Department of Computer Science and Information Engineering at the NCKU, Taiwan, in 2007. Currently, he is an assistant professor of the Department of Computer Science and Information Engineering, CJCU, Taiwan. His research interests include mobile/wireless/ spatio-temporal data management, sensor networks, skyline query processing, and web information retrieval.

Pei-Chi Liu received her ME degree in the Department of Computer Science and Information Engineering at the NCKU, Taiwan, in 2011. Currently, she is an associated researcher in the Department of Cloud Computing Laboratory at the Chunghwa Telecom. Her research interests include databases, mobile computing, and cloud computing.

Chiang Lee received his PhD degrees in electrical engineering from the University of Florida, Gainesville, in 1989. He joined IBM Mid-Hudson Laboratories, Kingston, New York, in 1989. He joined the faculty of National Cheng-Kung University in 1990 and is currently a professor of the Department of Computer Science and Information Engineering. His research interests are mobile/ sensor data management, bioinformatics, and integration of databases.