Optimal-Location-Selection Query Processing in Spatial Databases

Yunjun Gao, *Member*, *IEEE*, Baihua Zheng, *Member*, *IEEE*, Gencai Chen, and Qing Li, *Senior Member*, *IEEE*

Abstract—This paper introduces and solves a novel type of spatial queries, namely, *Optimal-Location-Selection* (OLS) search, which has many applications in real life. Given a data object set D_A, a target object set D_B, a spatial region R, and a critical distance d_c in a multidimensional space, an OLS query retrieves those target objects in D_B that are *outside* R but have *maximal optimality*. Here, the optimality of a target object $b \in D_B$ located outside R is defined as the number of the data objects from D_A that are *inside* R and meanwhile have their distances to b *not exceeding* d_c. When there is a tie, the *accumulated distance* from the data objects to b serves as the tie breaker, and the one with *smaller* distance has the *better optimality*. In this paper, we present the optimality metric, formalize the OLS query, and propose several algorithms for processing OLS queries efficiently. A comprehensive experimental evaluation has been conducted using both real and synthetic data sets to demonstrate the efficiency and effectiveness of the proposed algorithms.

Index Terms—Query processing, optimal-location-selection, spatial database, algorithm.

1 Introduction

Spatial databases play an important role in many real applications, including geographical information systems, decision support, intelligent transportation, and resource allocation. The key characteristic that makes a spatial database become a powerful tool is its ability to manipulate (e.g., model, index, and query, etc.), but not simply store, spatial data objects (e.g., points, line segments, rectangles, and polygons, etc.).

1.1 Motivation

Over the last decade, efficient query processing for spatial data objects has received considerable attention from the database research community. Representative spatial queries include range query [26], nearest neighbor (NN) search [5], [17], [33], spatial join [4], [21], [27], [28], and closest pair query [6], [7], [16]. However, there are still some interesting applications involving spatial data for decision making that cannot be efficiently supported by existing spatial query processing techniques. Consider the following two example applications.

Optimal watch point selection. Bird-watching is a popular activity in Singapore. In order to further promote this activity, the National Parks Board (NPB) decides to recommend some *optimal watch points*, so that bird-watchers can observe clearly as many bird species as possible at those recommended watch points. Let R be a bird habitat for bird-watching, D_A be a set of the locations l_i within R such that certain bird species normally appear, d_c be the maximal distance of the objects that can be viewed clearly through binoculars, and D_B be a set of potential watch points p_j outside R (i.e., $\forall p_j \in D_B, p_j \notin R$). Obviously, the more the bird species that can be observed clearly at one watch point, the better the watch point is. Hence, the *optimality* of a watch point $p_j \notin R$ can be quantified by $|\{l_i | l_i \in D_A \wedge l_i \in R \wedge dist(l_i, p_j) \leq d_c\}|$, where, without loss of generality, $dist(x, y)$ is a distance function that computes the euclidean distance between any two objects x and y, and $|S|$ is the cardinality of a set S.

Optimal lifeguard station selection. To ensure public safety, a limited number of lifeguards have to be strategically stationed on some of the busiest ocean beaches in the world. Locations of lifeguard stations should be close to as many accident-prone areas as possible. Let R be the water region which lifeguards are responsible for, D_A be a set of accident-prone regions $r_i \in R$ within an ocean beach, d_c be the maximum response time allowed, and D_B be a set of potential lifeguard locations l_j located outside R (i.e., $\forall l_j \in D_B, l_j \notin R$). Then, the *optimality* of a lifeguard station location l_j can be qualified by $|\{r_i | r_i \in D_A \wedge r_i \in R \wedge \frac{dist(r_i, l_j)}{v} \leq d_c\}|$, in which v denotes the average running velocity of lifeguards.

1.2 Problem Formulation

In light of the above applications, we, in this paper, introduce and solve a novel form of spatial queries, namely, *Optimal-Location-Selection* (OLS) search, which returns the object with

- *Y. Gao is with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore, and the College of Computer Science, Zhejiang University, Hangzhou 310027, PR China. E-mail: yjgao@smu.edu.sg, gaoyj@zju.edu.cn.*
- *B. Zheng is with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902, Singapore. E-mail: bhzheng@smu.edu.sg.*
- *G. Chen is with the College of Computer Science, Zhejiang University, Hangzhou 310027, PR China. E-mail: chengc@zju.edu.cn.*
- *Q. Li is with the Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong, PR China. E-mail: itqli@cityu.edu.hk.*

Manuscript received 25 Mar. 2008; revised 6 Oct. 2008; accepted 25 Mar. 2009; published online 31 Mar. 2009.
Recommended for acceptance by D. Srivastava.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2008-03-0161.
Digital Object Identifier no. 10.1109/TKDE.2009.81.

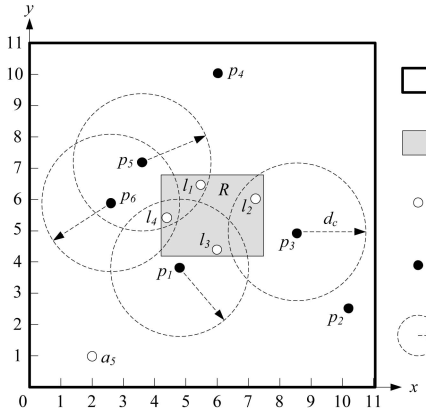

Fig. 1. Illustration of optimal watch point selection.

the *maximal optimality*. Before we present the optimality metric and formalize the OLS query, we first explain the intuition behind. Take the *optimal watch point selection* depicted in Fig. 1 as an example. For a specified watch point $p_j \notin R$, only those locations $l_i \in D_A$ with their distances to p_j not exceeding d_c can be viewed clearly. We assume that all these locations constitute p_j's *optimal set* S_{p_j}, defined in Definition 1. Since bird-watchers prefer to spot as many bird species as possible, the cardinality of S_{p_j} (i.e., $|S_{p_j}|$) has a direct impact on the optimality of the watch point p_j. The lager the $|S_{p_j}|$ is, the more optimal the watch point p_j is. In Fig. 1, for instance, the optimality of p_1 is better than that of p_3 as $|S_{p_1}| = |\{l_3, l_4\}| = 2$ and $|S_{p_3}| = |\{l_2\}| = 1$.

Definition 1: Optimal set. *Given a data object set D_A, a target object set D_B, a spatial region R, and a critical distance d_c, an* optimal set *S_{p_j} of a target object $b_j \in D_B$ that locates outside R is formed by all the objects $a_i \in D_A$ within R whose distances to b_j do not exceed d_c, i.e., $S_{b_j} = \{a_i | a_i \in D_A \wedge a_i \in R \wedge dist(a_i, b_j) \leq d_c\}$.*

However, when the optimal sets of two watch points share the same cardinality (e.g., $|S_{p_1}| = |S_{p_5}|$ in Fig. 1), the tie has to be broken. In general, bird-watchers prefer the location closer to them because they can spot bird species more easily. Thus, we employ the *accumulated distance* from the watch point p to all the locations l_i in its optimal set S_p, denoted as D_p $(= \sum_{l_i \in S_p} dist(l_i, p))$, as the tie breaker. The smaller the D_p is, the better the p's optimality is. For example, as shown in Fig. 1, $D_{p_1} = dist(l_3, p_1) + dist(l_4, p_1) = 1.1 + 1.6 = 2.7$ and $D_{p_5} = dist(l_4, p_5) + dist(l_1, p_5) = 1.8 + 1.9 = 3.7$. Although $|S_{p_1}| = |S_{p_5}| = 2$, the optimality of p_1 is better than that of p_5, as $D_{p_1} < D_{p_5}$ holds.

In summary, given a data object set $D_A = \{a_1, a_2, \ldots, a_n\}$, a target object set $D_B = \{b_1, b_2, \ldots, b_m\}$, a spatial region R, and a critical distance d_c in a multidimensional space, the optimality of a target object $b_j \in D_B$ located outside R is formally defined in Definition 2.

Definition 2: Optimality. *Given D_A, D_B, R, and d_c, the* optimality *of a target object $b_j \in D_B$ that is outside R, denoted by $b_j.OPT$, is defined as follows:*

$$b_j.OPT = |S_{b_j}| - \alpha_j, \tag{1}$$

where

$$\alpha_j = \frac{D_{b_j}}{d_c \times |S_{b_j}| + 1} = \frac{\sum_{a_i \in S_{b_j}} dist(a_i, b_j)}{d_c \times |S_{b_j}| + 1}.$$

The optimality metric considers not only the cardinality of b_j's optimal set S_{b_j} (i.e., $|S_{b_j}|$) but also the accumulated distance D_{b_j} from b_j to all the data objects included in S_{b_j}. We quantify the object optimality in such a way that it can be represented and ordered by a *single* value. Note that $\alpha_j = \frac{D_{b_j}}{d_c \times |S_{b_j}|+1} \in [0, 1)$, meaning that α value may change the ranking of objects in terms of optimality only when two objects have their optimal sets with the same cardinality. Take watch points p_1 and p_3 in Fig. 1 as an example. Since $S_{p_1} = \{l_3, l_4\}, S_{p_3} = \{l_2\}, p_1.OPT$ (>1) is for sure larger than $p_3.OPT$ (<1) no matter how large the value of α_1/α_3 is. In other words, as the optimal sets S_{p_1} and S_{p_3} have different cardinalities, the accumulated distances actually have *no* impact on their rankings in terms of optimality.

Based on the metric of optimality, there are three relationships between any two target objects $b_j, b_j' \in D_B$ that locate outside R, as stated in Definition 3, Definition 4, and Definition 5, respectively. The OLS query and the *k-optimal-location-selection* (*k*-OLS) query are developed to find the objects with the highest optimality, as formulated in Definition 6 and Definition 7, respectively.

Definition 3: $b_j \succ b_j'$. *Given $b_j, b_j' \in D_B$ outside R, b_j is* superior *to b_j', denoted by $b_j \succ b_j'$, if and only if $b_j.OPT > b_j'.OPT$.*

Definition 4: $b_j \equiv b_j'$. *Given $b_j, b_j' \in D_B$ outside R, b_j is* equivalent *to b_j', denoted by $b_j \equiv b_j'$, if and only if $b_j.OPT = b_j'.OPT$.*

Definition 5: $b_j \prec b_j'$. *Given $b_j, b_j' \in D_B$ outside R, b_j is* inferior *to b_j', denoted by $b_j \prec b_j'$, if and only if $b_j.OPT < b_j'.OPT$.*

Definition 6: Optimal-Location-Selection query. *Given D_A, D_B, R, and d_c in a multidimensional space, an OLS query returns the target object $b_j \in D_B$ having the* maximal optimality *among all the target objects that are outside R, i.e., $\forall b_j' \in D_B \wedge b_j' \notin R, b_j \succ\equiv b_j'$.*

Definition 7: *k*-Optimal-Location-Selection query. *Given D_A, D_B, R, d_c, and an integer k (≥1) in a multidimensional space, a* k-optimal-location-selection *(k-OLS) query retrieves the set of target objects, denoted as Res, such that: 1) Res contains k target objects from D_B, 2) $\forall b_j \in Res, b_j$ locates outside R (i.e., $b_j \notin R$), and 3)* none *of the nonresult object b_j' outside R is superior to any target object in Res, i.e., $\forall b_j \in Res$ and $\forall b_j' \in (D_B - Res) \wedge b_j' \notin R, b_j \succ\equiv b_j'$.*

1.3 Contributions

A naive solution to answer OLS (*k*-OLS) queries is to derive the optimality of each target object in D_B, and then, locate these *k* objects with the highest optimality. In the rest of this paper, we refer to this solution as a *baseline* approach. To be more specific, the baseline method adopts a filtering and refinement framework. In the filtering step, it prunes away all target objects in D_B with *zero* optimality, i.e., those target objects with their optimal sets being *empty*, and forms a candidate set C by including only those target objects $b_j \in D_B$ that are outside R, and meanwhile, have their minimal distances to R not exceeding d_c, i.e., $C = \{b_j | b_j \in D_B \wedge b_j \notin R \wedge \mathsf{mindist}(b_j, R) \leq d_c\}$. Thereafter, in the second refinement step, it derives the optimality for each candidate object $c \in C$ according to (1) and returns those k objects with the highest optimality.

The baseline approach is simple and straightforward, but it has two deficiencies. First, the cardinality of the candidate set C (i.e., $|C|$), which depends on the distribution of D_B and the sizes of d_c and R, might be very *large*. Consequently, a large number of target objects in C have to be evaluated, although the users may be only interested in the optimal one. Second, in order to derive the optimality of each candidate object c, it needs to traverse the data object set D_A once to retrieve the optimal set for the candidate object c (i.e., S_c). Hence, *multiple* traversals on D_A are incurred, resulting in high I/O overhead and expensive CPU cost. The poor performance of this baseline approach will be further demonstrated by our experiments, to be presented in Section 4.

Motivated by the significance of OLS (k-OLS) queries and the lack of efficient algorithms, in this paper, we propose three efficient k-OLS query processing algorithms, namely, *Three-Step algorithm* (TS), *Reuse-Based algorithm* (RB), and *Reverse Reuse-Based algorithm* (RRB). Our methods assume that both the data objects and the target objects are indexed by R-trees [1], [15], and utilize reuse techniques to avoid multiple scans of the data sets. All three algorithms guarantee that the search can be completed via one *single* traversal of the data object set D_A and the target object set D_B, respectively, whereas they employ different heuristics to prevent unnecessary traversals. To sum up, this paper has made following main contributions:

- we develop the optimality metric and formalize the OLS query and its generalization (i.e., k-OLS query),
- we propose several algorithms, including TS, RB, and RRB, for processing k-OLS queries efficiently, and
- we conduct extensive experiments using both real and synthetic data sets to verify the efficiency and effectiveness of our proposed algorithms under various settings.

1.4 Organization of the Paper

The rest of this paper is organized as follows: Section 2 surveys the existing work related to the OLS query problem. Section 3 elaborates three efficient k-OLS query processing algorithms. Section 4 presents comprehensive performance evaluation and reports our findings. Finally, Section 5 concludes the paper with some directions for the future work.

2 Related Work

In this section, we first briefly overview the R-tree and algorithms for range and NN queries, and then, review NN query variants and distance join queries.

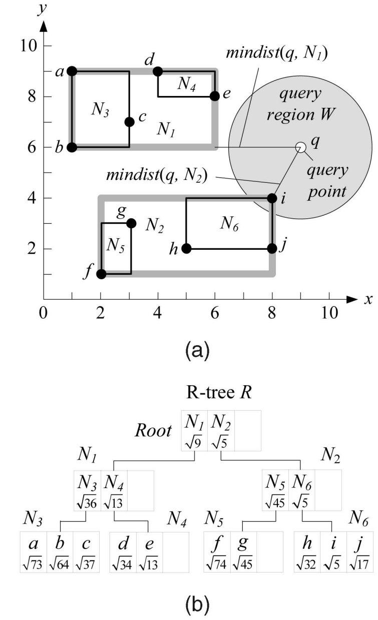

Fig. 2. Example of range and NN queries on the R-tree. (a) The point placement. (b) The corresponding R-tree.

2.1 Algorithms for Range and NN Queries Using R-Trees

Although our techniques can be used with other data-partitioning access methods (e.g., X-tree [2], etc.), we adopt R-tree as the underlying index structure due to its popularity. The R-tree [15] and its variants (e.g., the R*-tree [1]) are extensions of B-trees in a multidimensional space. Take the data set $\{a, b, \ldots, j\}$ depicted in Fig. 2 as an example, and we assume that each node accommodates up to three entries. R-tree groups points that are close to each other to form leaf nodes (e.g., points a, b, and c are grouped to form a leaf node N_3), and the leaf nodes are grouped together with the same principle to form intermediate (i.e., nonleaf) nodes. The clustering propagates until a *Root* node is formed. Each node corresponds to a *minimum bounding rectangle* (MBR) that bounds all its child entries. Normally, the search using R-tree tries to prune away nonqualified nodes/objects based on various distance metrics. For instance, metric $\mathsf{mindist}(q, N)$ indicates the minimal distance from a given query point q to a node N, and $\mathsf{mindist}(q, N)$ equals $dist(q, N)$ if N corresponds to a data point. The number in each entry N, as shown in Fig. 2b, represents $\mathsf{mindist}(q, N)$, which is not stored previously but computes *on-the-fly* during the query processing.

R-trees can efficiently support multiple spatial queries, including range queries and NN search. Range query returns the data objects in a given data set D that intersect or locate inside a specified query region W. For example, a range query issued at point q tries to find all the objects with their distances to q bounded by 2.5, as illustrated in Fig. 2a, where the shaded circle centered at q denotes the query region W. Starting from the root of the tree, the query is processed by recursively visiting those node entries whose MBRs intersect W. For instance, as $N_1 \cap W = \emptyset$, the subtree pointed by N_1 cannot contain any qualified object and the traversal of N_1 can be skipped. In contrast, node N_2 is accessed and its child nodes N_5 and N_6 are examined. The range query algorithm proceeds in the same manner until all the entries sharing common areas with W are visited and the final query result (i.e., point i) is returned.

Given a set D of objects and a query point q, an NN query finds the object in D that lies closest to q. Existing algorithms for NN queries on R-trees follow the *branch-and-bound* paradigm and utilize some distance metrics (e.g., mindist) to prune the search space, based on either *best-first* (BF) or *depth-first* (DF) traversal. The DF approach [5], [33] retrieves the NN(s) by traversing the R-tree in the depth-first fashion. As demonstrated in [31], the DF algorithm is *suboptimal*, i.e., it consumes more I/O than necessary.

The BF algorithm [17] achieves optimal I/O performance because it only accesses the nodes necessary for obtaining the NN. BF maintains a priority queue (e.g., a heap H) with the entries visited so far, sorted in ascending order of their mindist to q. First, it starts from the root of the tree and inserts all its child entries E into H together with mindist(E, q). As shown in Fig. 2, $H = \{(N_2, \sqrt{5}), (N_1, \sqrt{9})\}$. Then, the algorithm deheaps the top entry N_2 in H, accesses its child nodes, and enheaps all the entries, after which $H = \{(N_6, \sqrt{5}), (N_1, \sqrt{9}), (N_5, \sqrt{45})\}$. Similarly, it deheaps the next top entry, i.e., a leaf entry N_6, and adds its enclosed points (i.e., h, i, j) to H. At this time, heap H is updated to $\{(i, \sqrt{5}), (N_1, \sqrt{9}), (j, \sqrt{17}), (h, \sqrt{32}), (N_5, \sqrt{45})\}$. The next deheaped entry is a data point i. As it is the first data point discovered, point i is taken as the current NN. Since the next head entry (i.e., N_1) in H is farther (from q) than i, the search process is terminated and i is returned as the final query result. BF can be easily extended for the retrieval of k ($\geq$1) NNs. Furthermore, BF is *incremental*, that is, it reports the NNs in ascending order of their distances to q, so that k does not have to be known in advance.

2.2 NN Query Variants

In addition to conventional (i.e., point) NN search, many variations of NN queries have been proposed in the literature. *Reverse* NN (RNN) search [19], [39] retrieves all the data points that have a given query point as their NN. *Constrained* NN search [11] finds the NN(s) in a constrained region of the data space. *All* NN search [43] returns, for each data object $a_i \in D_A$, its NN $b_j \in D_B$. *Aggregate* NN (ANN) query [29], [30] retrieves the point(s) in a data set D_B (e.g., facilities) with the *smallest* aggregate distance(s) to points in another data set D_A (e.g., queries). For example, there are n users at locations $q_1, q_2, \ldots, q_n$, a *sum* ANN query finds the facility $f \in D_B$ that minimizes the summation of its distance to all the users, i.e., $\forall f'(\neq f) \in D_B, \sum_{i=1}^{n} dist(q_i, f) \leq \sum_{i=1}^{n} dist(q_i, f')$. Our proposed OLS query is similar to ANN query in that they both involve two data sets and consider the distances between objects. However, there are several differences. First, OLS query imposes a restricted region R (instead of the *whole data space*). Second, OLS query takes into account a distance threshold d_c (instead of *infinity*). Third, they employ different sorting criteria to order the result objects. The ANN query outputs answer objects according to a specified aggregate function (e.g., *sum*, *max*, *min*, etc.), while the OLS query returns answer objects based on the *optimality metric* (presented in Definition 2).

All the above work focuses on *snapshot* queries in which the evaluation of query happens only *once*. In order to cater for the applications that require continuous query evaluation, various types of continuous queries have been explored. *Continuous* NN (CNN) search [38], [40] is such a type. It aims at finding the NN for each point along a given query line segment. In addition, the CNN *monitoring* problem that monitors the answer objects to a CNN query for a specified duration has recently been studied, and efficient algorithms (e.g., CPM [22], SEA-CNN [41], and YPK-CNN [42]) have been proposed as well. Other versions of CNN monitoring include: 1) CNN monitoring in the distributed environment [23] and 2) CNN monitoring in the road network [24].

More recently, some other NN query variants, such as *surface* kNN query [9], *range* NN retrieval [18], and kNN search over moving object trajectories [12], [13], [14] have also been proposed and solved. However, to the best of our knowledge, this paper is the first piece of work aiming at efficiently handling OLS queries in spatial databases. It is worthwhile to point out that the proposed OLS query is different from the existing *Optimal-Location* (OL) query [10]. OL search uses a different metric to evaluate the object influence/optimality. Given a set S of sites and a set O of weighted objects, the influence of a specified site $l \in S$ is the *total weight* of the objects in O that take l as their NN. The OL query is to find the site l inside a given spatial region Q with the maximal influence. Thus, it has to locate the RNN objects of l in order to derive l's influence. Nevertheless, the OLS query quantifies the optimality of a specified site l based on the data objects that are physically close to l, and meanwhile, have their distances to l bounded by a predefined threshold d_c.

2.3 Distance Join Queries

Given two data sets D_A and D_B, the *distance join* operation is to compute and rank a subset of the Cartesian product of sets D_A and D_B based on a specified distance order. Since it was introduced by Hjaltason and Samet [16] in spatial databases, the distance join has received considerable attention, largely due to its importance in many data analysis tasks, e.g., data mining and clustering [25].

As mentioned earlier, distance join queries actually order the data object pairs from $D_A \times D_B$ according to different distance functions. Among different distance joins, *closest pair query* (CPQ) is one of the most popular. Given two spatial data sets D_A and D_B, the CPQ retrieves the pair of data objects (a, b) with $a \in D_A$ and $b \in D_B$, such that their distance is the smallest, i.e., $\forall (a', b') \in D_A \times D_B, dist(a, b) \leq dist(a', b')$. Based on the way that the result objects are delivered, the algorithms can be classified into two categories, namely, *incremental algorithms* and *nonincremental algorithms*.

Incremental algorithms, e.g., the one based on *best-first* traversal [16], report the result objects *one by one*. The key idea is to maintain a priority queue which contains pairs of node entries $(N_A, N_B) \in D_A \times D_B$, sorted in ascending order of their distances. The advantages of this approach are as follows: 1) k has not to be known in advance and the user can stop the algorithm when he/she is satisfied with the result and 2) the incremental algorithms outperform nonincremental methods when k is small, as demonstrated in [16]. Some techniques to enhance the search performance by using bidirectional node expansion, plane-sweeping, and adaptive multistage techniques have been proposed in [35], [36].

As k value increases, the pruning power of distance priority queue degrades, because the cutoff value (i.e., pruning distance) stored in the distance priority queue may remain high for a long duration. Accordingly, nonincremental algorithms are proposed to improve the pruning process based on several distance metrics such as minmindist, minmaxdist, and maxmaxdist. Different from incremental algorithms, nonincremental algorithms report the result objects as a whole at the end of the query processing. Consequently, the main issue that nonincremental algorithms have to address is how to separate the treatment of the terminal candidates (i.e., the elements of the final result) from the rest of the candidates. Example algorithms include a recursive *depth-first* algorithm proposed in [6] and an iterative *best-first* algorithm proposed in [7].

In addition, some variants of the distance join have been reported in the literature. Koudas and Sevcik [20] propose the *similarity* join (also called δ-distance join), which involves two spatial data sets and a distance threshold δ, and returns pairs of data objects within distance δ from each other. Bohm and Krebs [3] discuss the *k-nearest neighbor* join, which associates two sets of spatial data objects D_A and D_B and a cardinality threshold k; the output is a set of pairs from D_A and D_B that include, for each data object from D_A, its k NNs in D_B. Shou et al. [37] study the *iceberg distance join* where, given two spatial data sets D_A and D_B, a distance threshold δ, and a cardinality threshold k, the target is to retrieve all pairs of data objects from D_A and D_B such that: 1) the pairs of data objects are within distance δ from each other and 2) a data object of D_A appears at least k times in the final result. Corral et al. [8] introduce the *k-multiway distance join*, which involves n spatial data sets, a query graph QG (i.e., a weighted directed graph that defines directed itineraries between the n input data sets), and a cardinality threshold k; the answer is a set of k distinct n-tuples (i.e., tuples of n data objects from the n data sets obeying the QG) with the k smallest $D_{distance}$-value which is the value of a linear function of distances of the n data objects that constitute this n-tuple, according to the edges of the QG. Recently, the problem of processing distance join queries with spatial region constraints has also been investigated in [32], [34].

Like distance join, the OLS query involves two data sets D_A and D_B and evaluates the objects based on distance. However, they are fundamentally different. First, their result sets are different. OLS query aims at target data set D_B, while distance join retrieves data object pairs from $D_A \times D_B$. Second, they adopt different metrics. OLS query evaluates each target object $b_j \in D_B$ that is outside a given spatial region R according to our newly defined *optimality metric* (see Definition 2). This means, whether b_j is an answer object not only depends on the data objects in D_A that are inside R, and meanwhile, have their distances to b_j not exceeding a specified distance threshold d_c, but also depends on the optimality of other data objects. On the other hand, distance join employs a totally different ordering function. It retrieves the data object pair $(a, b) \in D_A \times D_B$ with minimum distance. Last but not the least, OLS query takes a spatial region R and a critical distance d_c as inputs, whereas the distance join may or may not consider any other parameters. Therefore, the OLS query differs from the distance join query, and we need new approaches to tackle it efficiently.

TABLE 1
Symbols and Description

Notation	Description
D_A	A set of data objects in a multi-dimensional space
D_B	A set of target objects in a multi-dimensional space
T_A	The R-tree on D_A
T_B	The R-tree on D_B
k	The number of required answer objects for an OLS query
R	A spatial region
d_c	A critical distance
a	A data object in D_A
$a.v_j$	The coordinate of object a along the j-th dimension
b	A target object in D_B
$b.OPT$	The optimality of any target object $b \in D_B$
S_b	The optimal set of any target object $b \in D_B$
H_t	The top/head entry of heap H
Res	The result set of an OLS query

Another work that is very related to OLS query is *Top-k Spatial Join* (TSJ) [44]. Given two data sets D_A and D_B, the TSJ retrieves the k objects from D_A or D_B that intersect the maximum number of objects from the other data set. Several efficient TSJ processing algorithms have been proposed in [44], and they, as mentioned in [44], can be easily extended to support *Top-k distance Semijoin* where, given two data sets D_A, D_B and a range e, the goal is to return the k objects in D_B that enclose the largest number of objects from D_A within the distance e. The OLS query is a specific form of TSJ, as it imposes a spatial region R to limit the number of objects from D_A that need consideration and utilizes the optimality metric to rank answer objects. These constraints might render the generic methods for TSJ processing inefficient, which motivates the work presented in this paper.

3 OLS Query Processing

In this section, we propose three algorithms for efficiently processing OLS queries, assuming that the data object set D_A and the target object set D_B are indexed by two separate R-trees. Table 1 lists the symbols used in the rest of this paper.

In order to facilitate the understanding of different OLS query processing algorithms, a running example, as depicted in Fig. 3a, is employed. Here, the data object set $D_A = \{a_1, a_2, \ldots, a_8\}$, as shown in Fig. 3b; the target object set $D_B = \{b_1, b_2, \ldots, b_8\}$, as shown in Fig. 3d; the shaded rectangle represents the spatial region R; and d_c is set to 2. We further assume that the number of required answer objects k is 1. The corresponding R-trees on D_A and D_B with node capacity set to two entries are illustrated in Figs. 3c and 3e, respectively.

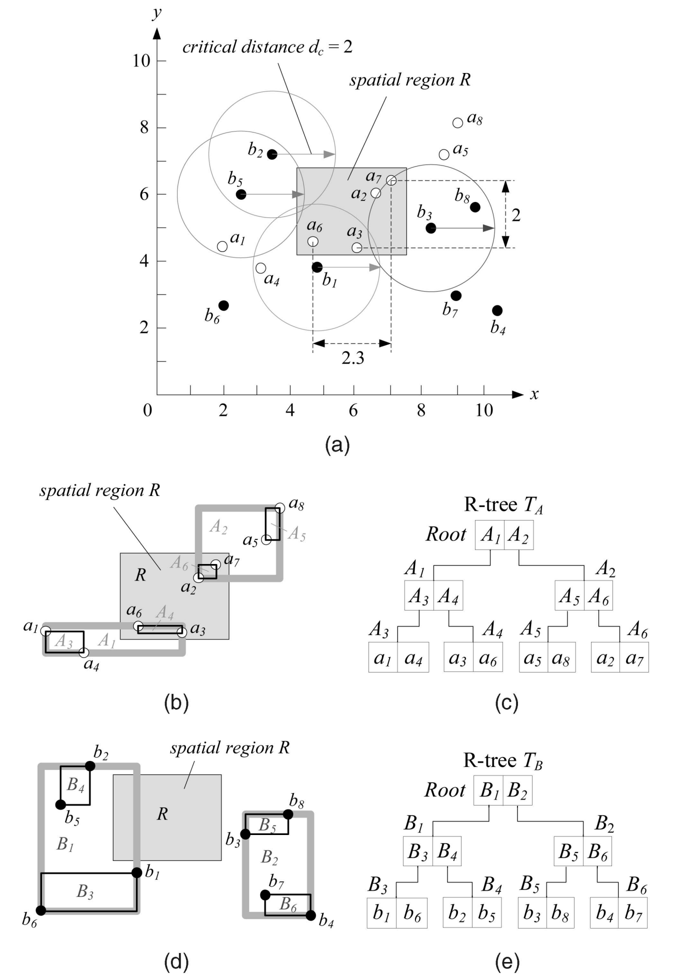

Fig. 3. A running example. (a) The data and target object placement. (b) Data object set D_A. (c) The R-tree T_A. (d) Target object set D_B. (e) The R-tree T_B.

3.1 Three-Step Algorithm

As explained in Section 1.3, the main issue of the baseline algorithm is that the data set D_A has to be traversed *multiple times*. To address this issue, a *Three-Step algorithm* (TS) is developed. It aims at visiting D_A only *once* in order to improve the search performance. More specifically, in step 1, all the data objects in D_A that are within the specified spatial region R are retrieved via a window/range query and preserved in a stack st_A. In step 2, D_B is accessed and all the target objects $b \in D_B$ that are outside R, and meanwhile, have their minimal distance to R not exceeding d_c, i.e., $\mathsf{mindist}(b, R) \leq d_c$, are retrieved and maintained in a stack st_B. Finally, in the third step, all the candidate target objects in st_B are evaluated, with those data objects affecting their optimality already available in st_A.

Without any auxiliary information, we need to access each object $a \in st_A$ to evaluate its impact on the optimality of a given object $b \in st_B$. Consequently, the time complexity of the last step is $O(|st_A| \cdot |st_B|)$. Actually, only those objects with their distances to b bounded by d_c can contribute to the optimality of b. In order to avoid the retrieval and evaluation of unnecessary objects from st_A, we strategically sort the objects in st_A/st_B based on their coordinate values along the

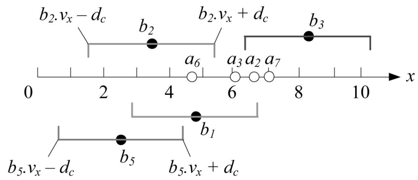

Fig. 4. Retrieval deduction via sorting.

dimension with *largest span*, denoted by $\hat{i}$. Let $\overline{cor_i}$ and $\underline{cor_i}$ represent the maximal and minimal coordinate values of those objects in st_A along the ith dimension, and the one with the largest span $\hat{i}$ will be the ith dimension such that $\forall j \neq i, \overline{cor_i} - \underline{cor_i} \geq \overline{cor_j} - \underline{cor_j}$. The main motivation behind this ordering is that if the distance between a data object a and a target object b along the dimension $\hat{i}$ is already larger than d_c, a for sure will not affect b's optimality, and hence, the retrieval and evaluation of a can be saved.

Back to our running example shown in Fig. 3a. As $st_A = \{a_6, a_3, a_2, a_7\}$ and the coordinate difference along the x-dimension is larger than that along the y-dimension, $\hat{i}$ is set to x. Thereafter, all the objects in st_A and st_B are sorted according to ascending order of their coordinates along x-dimension, as illustrated in Fig. 4. Here, the notation $\llcorner\!\_\!\lrcorner$ represents the search range of the optimal set S_b for a target object b. It is known that S_b only includes those objects that are inside a given spatial region R, and meanwhile, have their distances to b bounded by d_c. If $|b.v_{\hat{i}} - a.v_{\hat{i}}| > d_c$, it is guaranteed that $dist(b, a) > d_c$, and thus, a for sure will be excluded from S_b. Take object b_2 as an example. Without sorting, it has to check all the four objects in st_A to form S_{b_2}. However, with data objects sorted according to their coordinates along x-dimension, only object a_6 with its x-coordinate value fallen into the range $[b_2.v_x - 2, b_2.v_x + 2]$, i.e., $(b_2.v_x - 2) \leq a_6.v_x \leq (b_2.v_x + 2)$, needs evaluation.

The pseudocode of TS algorithm is presented in Algorithm 1. TS first initializes a min-heap H as a temporary storage of the result, a min-heap H_A for keeping all the objects that will be included into the optimal set S_b of a target object b, and two stacks st_A and st_B (line 1). It then starts the first step, i.e., finding all the objects from D_A that locate inside the specified spatial region R via a window query on T_A, and ordering them based on coordinate values along the dimension $\hat{i}$ with maximal span (lines 2-3). Similarly, the target objects that are outside R and have their minimal distances to R not exceeding d_c are retrieved from D_B and maintained in st_B, again via a window query on T_B (lines 4-5). Thereafter, each candidate in st_B is evaluated (lines 6-12) and those k candidates with best optimality are returned as the final query result (line 13).

Algorithm 1. Three-Step Algorithm (TS)
Input: T_A, T_B, k, R, d_c;
Output: Res;
Procedure:
1: initialize two min-heaps $H = H_A = \emptyset$ and two stacks $st_A = st_B = \emptyset$;
2: $T = \{a | a \in D_A \wedge a \in R\}$ and decide $\hat{i}$;

3: sort objects in T according to ascending order of their coordinates along $\hat{i}$-th dimension and put them in st_A;
4: $T = \{b | b \in D_B \wedge b \notin R \wedge \mathsf{mindist}(b, R) \leq d_c\}$;
5: sort objects in T according to ascending order of their coordinates along $\hat{i}$-th dimension and put them in st_B;
6: **for** each point $b \in st_B$ **do**
7: $\quad P = \{a \in st_A \wedge a.v_{\hat{i}} \in [b.v_{\hat{i}} - d_c, b.v_{\hat{i}} + d_c]\}$ and $H_A = \emptyset$;
8: $\quad$ **for** each point $a \in P$ **do**
9: $\quad\quad$ **if** $dist(a, b) \leq d_c$ **then**
10: $\quad\quad\quad$ insert $(a, dist(a, b))$ into H_A;
11: $\quad$ calculate $b.OPT$ using Equation (1);
12: $\quad$ ResultUpdate $(H, b, b.OPT)$;
13: $Res = H$ and return Res;

Function: ResultUpdate $(H, b, b.OPT)$
14: **if** $|H| < k$ **then**
15: $\quad$ insert $(b, b.OPT)$ into H;
16: **elsa if** $|H| = k$ and $H_t.OPT < b.OPT$ **then**
17: $\quad$ remove H_t entry from H;
18: $\quad$ insert $(b, b.OPT)$ into H;

Back to the running example. In step 1, $st_A = \{a_6, a_3, a_2, a_7\}$ is formed, with $\hat{i}$ set to x. In step 2, $st_B = \{b_5, b_2, b_1, b_3\}$ is formed. Subsequently, in step 3, the optimal set of each target object $b \in st_B$ is formed by scanning those objects in st_A with x-coordinate values fallen inside the range $[b.v_x - d_c, b.v_x + d_c]$, based on what its optimality is derived. For instance, the optimal set for $b_2 \in st_B$ is $\emptyset$ and its optimality $b_2.OPT = 0$. As for $b_1 \in st_B$, its optimal set is $\{a_6, a_3\}$ and its optimality $b_1.OPT = |S_{b_1}| - \frac{dist(a_6,b_1)+dist(a_3,b_1)}{d_c \times 2+1} \approx 1.42$. Finally, the target object with the highest optimality (i.e., b_1) is returned as the result.

Let $|T_A|$ and $|T_B|$ be the tree size of T_A and T_B, respectively, $|st_A|$ and $|st_B|$ be the cardinality of stack st_A and stack st_B, respectively, and $|D_A|$ and $|D_B|$ be the size of D_A and D_B, respectively. Without loss of generality, we assume that R is *much smaller* than the original (i.e., whole) data space, $|st_A| < |D_A|$ and $|st_B| < |D_B|$. Then, Lemma 1 analyzes the time complexity of TS and Lemma 2 proves its correctness.

Lemma 1. *The time complexity of the TS algorithm is* $O(log|T_A| + log|T_B| + |st_A| \times |st_B|)$.

Proof. The TS algorithm follows the three-step framework. In the first stage, TS takes $O(log|T_A|)$ to find all the data objects from D_A that are inside R; in the second stage, it incurs $O(log|T_B|)$ to retrieve all the candidate target objects; and in the third stage, it takes $O(|st_A| \times |st_B|)$ to evaluate the optimality of each candidate target object. Hence, the total time complexity of the TS algorithm is $O(log|T_A| + log|T_B| + |st_A| \times |st_B|)$. □

Lemma 2. *The TS algorithm reports* exactly *the top-k target object(s) that have the maximal optimality among all the target objects that locate outside R.*

Proof. During the processing of TS, it retrieves all the target objects $b \in D_B$ that are outside R, and meanwhile, have their smallest distances to R bounded by d_c as candidate answer objects, and thus, no answer objects are missed (i.e., no false negatives). In the subsequent step, TS evaluates every candidate by considering the impact of the data objects from D_A that are inside R, which ensures no false positives. Consequently, the correctness of the TS algorithm is guaranteed. □

3.2 Reuse-Based Algorithm

As mentioned before, the main issue of the baseline algorithm is that it needs to scan data set D_A multiple times. TS algorithm tackles this issue by fetching all the objects (from D_A) inside R via traversing T_A once and maintaining them in a stack st_A. However, the access to an object $a \in st_A$ is necessary only when it affects at least one target object's optimality, i.e., $\exists b \in D_B$ such that $a \in S_b$. Thus, blindly fetching all the objects within R to form st_A is not always the best choice, especially when the given spatial region R is a hot area with st_A containing a large number of objects but actually many objects do not contribute to any optimal set corresponding to a target object. Motivated by this observation, we propose an alternative, namely, *Reuse-Based algorithm* (RB). The main idea is to trigger the access of nodes/points in T_A via the examination of target objects. Specifically, we evaluate candidate target objects one by one, according to the ascending order of their minimal distances to R. When a candidate target object $b \in D_B$ is evaluated, we traverse the data set D_A to find out all the data objects that can contribute to the optimal set of b (i.e., S_b). Since the same data objects may contribute to the optimal sets of different target objects, stacks $st/temp$ are employed to keep track of the current view of D_A in order to enable reuse. Therefore, we access the nodes in st and expand the node only when it is very likely to contain objects that will be included into the optimal set of the current evaluated target object.

The pseudocode of RB algorithm is shown in Algorithm 2. RB first initializes a min-heap H to store temporary results, two min-heaps H_A and H_B to accommodate the optimal set S_b for a specified target object b and the candidate target object set, respectively, and two stacks st and $temp$ to keep track of the nodes/points of T_A visited so far (line 1). Then, it retrieves the candidate set by issuing a window query based on T_B (line 2). Thereafter, it recursively evaluates every candidate until H_B is empty (lines 4-10).

Algorithm 2. Reuse-Based Algorithm (RB)
Input: T_A, T_B, k, R, d_c;
Output: Res;
Procedure:
1: initialize three min-heaps $H = H_A = H_B = \emptyset$ and two stacks $st = temp = \emptyset$;
2: $H_B = \{b | b \in D_B \wedge b \notin R \wedge \mathsf{mindist}(b, R) \leq d_c\}$;
3: push $(T_A.root, 0)$ into st;
4: **while** $H_B \neq \emptyset$ **do**
5: $\quad$ de-heap the top entry b from H_B;
6: $\quad$ $H_A = \emptyset$;
7: $\quad$ Traverse-D_A-Reuse $(T_A, b, R, d_c, st, temp, H_A)$;
8: $\quad$ calculate $b.OPT$ according to Equation (1);
9: $\quad$ ResultUpdate $(H, b, b.OPT)$;
10: $\quad$ $st = temp \cup st$ and $temp = \emptyset$;
11: $Res = H$ and return Res;

```
Function: Traverse-D_A-Reuse (T_A, b, R, d_c, st, temp, H_A)
12: while st ≠ ∅ do
13:   pop the top entry e out of st;
14:   if e is a point then
15:     push e into temp;
16:     if dist(e, b) ≤ d_c then
17:       insert (e, dist(e, b)) into H_A;
18:   else
19:     for each child entry e_i ∈ e do
20:       e'_i = e ∩ R;
21:       if midist(e'_i, b) ≤ d_c then
22:         push (e'_i, midist(e'_i, b)) into st;
23:       else
24:         push e'_i into temp;
```

As the optimality of a target object b is dependent on its optimal set S_b, D_A has to be scanned for obtaining S_b. In order to enable reuse of previously accessed nodes/points from D_A, two stacks st and $temp$ are employed. Stack st with initial value set to $T_A.root$ (i.e., the root of T_A) guides the traversal of T_A. Since the purpose of accessing T_A is to form the optimal set S_b, it only accesses those nodes that might contribute to S_b, while the others are preserved in $temp$ for the formation of the optimal sets of other target objects. Once the optimal set S_b is formed, b's optimality $b.OPT$ is derived, and the result set is updated if necessary (lines 8-9). Thereafter, st is set to $temp \cup st$ in order to enable node/point reuse and to get ready for the evaluation of the next target object (line 10).

The formation of the optimal set S_b for a given target object b is handled by the Function Traverse-D_A-Reuse (lines 12-24). It visits the nodes/points e locally available (i.e., $e \in st$) and the detailed evaluation of e depends on its type. If e is a data point, it is checked against the spatial region R and its distance to b is derived. If e is qualified, i.e., $dist(e, b) \leq d_c \wedge e \in R$, it is inserted into H_A (lines 14-17). Otherwise, e must be a nonleaf node and we need to access its child entries via scanning T_A. As only those objects $a \in D_A$ that are inside R will contribute to some optimal sets corresponding to target objects, we only consider the portion of the entry that is within R, denoted as e'_i (line 20). If e'_i for sure will not contribute to S_b (i.e., $\mathsf{mindist}(e'_i, b) > d_c$), it is inserted into $temp$ (lines 23-24). Otherwise, it is inserted into st (lines 21-22). The Traverse-D_A-Reuse function terminates when all the entries in the stack st are accessed. Then, we can derive the optimality of b (i.e., $b.OPT$) based on H_A, which maintains b's optimal set S_b, and update the result set if necessary.

Consider our example again. After initialization, RB retrieves all the potential target objects to form $H_B = \{b_1, b_3, b_2, b_5\}$, and then, starts the evaluation of candidate target objects in H_B. First, b_1 with the smallest $\mathsf{mindist}(b_1, R)$ is evaluated. As it is the first evaluated target object, we start traversing T_A from the root node. Based on R and d_c, node A_1 that has $\mathsf{mindist}(b_1, A_1) = 0 < d_c$ is pushed to st, while A_2 is preserved in $temp$. Then, A_1 is expanded and both its child nodes A_3 and A_4 are accessed. Since A_3 locates outside R completely, it is discarded; whereas, A_4 is pushed to st for later evaluation as $\mathsf{mindist}(b_1, A_4) < d_c$. Next, A_4 is visited and its child points a_6 and a_3 are inserted into H_A (i.e., the storage of the optimal set S_{b_1}) because both have their distances to b_1 not exceeding d_c. After evaluating A_4, the stack st becomes empty, and hence, the formation of b_1's optimal set S_{b_1} is completed, with $temp = \{a_6, a_3, A'_2 = A_2 \cap R\}$.

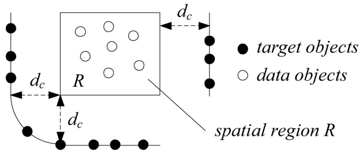

Fig. 5. Overestimated candidate set.

Thereafter, target object b_3 is evaluated. RB first accesses A'_2 and pushes A_6 into st. Then, it visits the child points of A_6, i.e., a_2 and a_7, and inserts them into H_A to form the optimal set S_{b_3}. The search finishes as the stack st is empty, with $temp = \{a_6, a_3, a_2, a_7\}$. The next target object evaluated is b_2. According to locally available nodes/points, its optimal set S_{b_2} is *empty*, and thus, the access to T_A is saved. Similarly, upon the evaluation of b_5, all the locally available nodes/points have their minimal distances to b_5 exceeding d_c, and hence, $S_{b_5} = \emptyset$. Now, H_B is empty and b_1 is returned as the final query result to complete the search. This example shows a worst case scenario for the RB algorithm as all the objects inside the specified spatial region R are accessed. However, the accesses to some data objects might be saved in some other cases. For example, when d_c is set to *one* but not *two* for the same example, RB only visits *two* out of *four* data objects (i.e., a_6 and a_3), while previous TS algorithm has to access all four objects.

Let α indicate the maximal number of node accesses incurred during the evaluation of a target object, $|H_B|$ be the size of heap H_B, and $|st|$ and $|temp|$ be the cardinality of stacks st and $temp$, respectively. Lemma 3 presents the time complexity of RB algorithm. Note that we ignore the proof of Lemma 3, which is very similar to that of Lemma 1.

Lemma 3. *The time complexity of the RB algorithm is* $O(log|T_B| + |H_B| \times log(|st| + |temp| + \alpha))$.

3.3 Reverse Reuse-Based Algorithm

Both TS algorithm and RB algorithm make a conservative assumption, i.e., all the target objects with their minimal distances to a given spatial region R not exceeding d_c have the potential to become part of the result. Consequently, they retrieve all the target objects b that are outside R and have $\mathsf{mindist}(b, R) \leq d_c$ to constitute a candidate set C. However, only those target objects b with *nonempty* optimal set (i.e., $S_b \neq \emptyset$), but definitely not others, may become the answer objects for an OLS query.

Take an extreme case where none of the objects in C is qualified for the OLS query as an example. As shown in Fig. 5, all the target objects, represented by dot points, have their minimal distances to R equivalent to d_c, while the data objects, denoted by hollow points, are close to the center, but not the boundary, of the spatial region R. For any target object b, there is no data object a such that $a \in R \wedge dist(a, b) \leq d_c$. In other words, the optimal set S_b for $\forall b \in C$ is empty. If the distribution of the data objects inside R is known in advance, the access and evaluation of some target objects that for sure have empty optimal sets could be saved. Motivated by this observation, we propose our third algorithm, namely, *Reverse Reuse-Based algorithm* (RRB). Here, we use the term "reverse" to distinguish RRB from the previous RB algorithm in which all the potential target objects are accessed to guide the retrieval of data objects. Differently, the RRB algorithm fetches all the data objects inside the spatial region R to guide the access of target objects.

Algorithm 3 depicts the pseudocode of RRB algorithm. RRB first initializes a min-heap H to accommodate temporary results, a min-heap H_A to keep all the data objects that locate inside a given spatial region R, three stacks st_B, st, and $temp$ with st_B holding all the target objects b whose optimality may be affected by a specified data object a, and st and $temp$ serving as the working stack and the auxiliary stack, respectively, as mentioned in previous RB algorithm (line 1). In addition, it also maintains a list, *InfoList*, with its functionality explained later. Once the initialization is done, RRB retrieves all the data objects from D_A that are within R and maintains them in H_A (line 2). Thereafter, it deheaps the head entry of H_A for evaluation until H_A is empty (lines 4-10).

Algorithm 3. Reverse Reuse-Based Algorithm (RRB)
Input: T_A, T_B, k, R, d_c;
Output: Res;
Procedure:
1: initialize two min-heaps $H = H_A = \emptyset$, three stacks $st_B = st = temp = \emptyset$, and $InfoList = \emptyset$;
2: $H_A = \{a | a \in D_A \wedge a \in R\}$;
3: push $(T_B.root, 0)$ into st;
4: **while** $H_A \neq \emptyset$ **do**
5: de-heap the top entry a from H_A;
6: Traverse-D_B-Reuse $(T_B, a, R, d_c, st, temp, st_B)$;
7: **while** $st_B \neq \emptyset$ **do**
8: pop the head entry $(b, dist(b, a))$ out of st_B;
9: append $(a, dist(b, a))$ into the list associated with b;
10: $st = temp \cup st$ and $temp = \emptyset$;
11: **for** each b in *InfoList* **do**
12: calculate $b.OPT$ according to Equation (1);
13: ResultUpdate $(H, b, b.OPT)$;
14: $Res = H$ and return Res;

Function: Traverse-D_B-Reuse $(T_B, a, R, d_c, st, temp, st_B)$
15: **while** $st \neq \emptyset$ **do**
16: pop the top entry e out of st;
17: **if** e is a point **then**
18: push e into $temp$;
19: **if** $dist(e, a) \leq d_c$ **then**
20: push $(e, dist(e, a))$ into st_B;
21: **else**
22: **for** each child entry $e_i \in e$ **do**
23: **if** e_i is not completely inside R **then**
24: **if** midist$(e_i, a) \leq d_c$ **then**
25: push $(e_i$, midist$(e_i, a))$ into st;
26: **else if** mindist$(e_i, R) \leq d_c$ **then**
27: push $(e_i$, midist$(e_i, a))$ into $temp$;

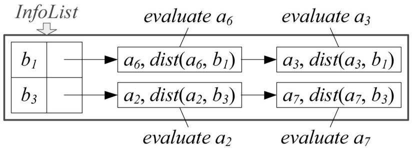

Fig.6. Example of *InfoList*.

The main idea is that for each data object a, we find out those target objects b whose optimality will be affected by a, i.e., $\{b | b \in D_B \wedge b \notin R \wedge dist(a, b) \leq d_c\}$. As the optimality of a target object b cannot be derived unless all the data objects included into its optimal set S_b are retrieved, we maintain the partial optimal set (i.e., the fact that object a is contained in S_b) in *InfoList*. Each list corresponds to one target object b, with all the data objects a accessed so far that will contribute to b's optimality appended, in the format of $(a, dist(a, b))$. After all the data objects $a \in H_A$ are evaluated, the formation of the optimal sets for all the candidate target objects is completed. Then, the optimality of each candidate can be derived and the result set can be obtained (lines 11-13). Finally, the algorithm outputs the result (line 14).

Different from previous RB algorithm, the reuse technique is applied to the data set D_B but not D_A. As shown in Function Traverse-D_B-Reuse (lines 15-27), the access of D_B (i.e., T_B) is triggered by the data object a. Whenever a new data object a is retrieved, the stack st is scanned to find out those target objects b satisfying $dist(a, b) \leq d_c$.

Again, back to our running example. RRB first fetches all the data objects located inside R to form $H_A = \{a_6, a_3, a_2, a_7\}$, and then, evaluates objects in H_A *one by one*. When $a_6 \in H_A$ is evaluated, it triggers the access of node B_1 and preserves B_2 in $temp$ due to the fact that mindist$(B_2, a_6) > d_c$ and mindist$(B_2, R) < d_c$. B_1 has two child entries B_3 and B_4. B_4 is kept in $temp$ as mindist$(B_4, a_6) > d_c$ but mindist$(B_4, R) < d_c$. On the other hand, B_3 is pushed into st as its distance to a_6 is smaller than d_c. Thereafter, B_3's child entries b_1 and b_6 are accessed. The optimality of b_1 is affected by a_6, and hence, b_1 is added to st_B and *InfoList*, while b_6 is discarded because its distance to R exceeds d_c. Now, the st is empty and the evaluation of a_6 is finished with $temp = \{b_1, B_4, B_2\}$ and $st_B = \{(b_1, dist(b_1, a_6))\}$.

Then, $a_3 \in H_A$ is evaluated. Based on locally available nodes/points, it only affects b_1's optimality, and thus, the *InfoList* is updated. Next, $a_2 \in H_A$ is evaluated. As mindist$(a_2, B_2) < d_c$, it triggers the access of node B_2, pushes B_2's child entry B_5 into st, and preserves the other entry B_6 in $temp$. When B_5 is visited, its child entry b_3 is added to st_B and *InfoList*, whereas b_8 is discarded since mindist$(b_8, R) > d_c$. At this time, the evaluation of a_2 terminates, and $temp$ is updated to $\{b_3, b_1, B_4, B_6\}$. Finally, $a_7 \in H_A$ is evaluated. Based on local information, it only affects the optimality of b_3. Consequently, it does not trigger the access of any node and the evaluation is completed, with *InfoList* updated. The final *InfoList* situation is illustrated in Fig. 6, based on which the optimality of target objects is derived. Object b_1 is returned as the result of the OLS query. Compared with

TABLE 2
Description of Real Data Sets

Dataset	Description
CL	9, 203 2D cultural landmarks in North America
LB	53, 143 2D points in Long Beach County
CA	62, 173 2D points in California
RS	191, 558 2D railroad segments in North America

the TS algorithm, the access of target objects b_2 and b_5 is saved.

Let β denote the maximal number of node accesses required during the evaluation of a data object, $|H_A|$ be the size of heap H_A, and $|InfoList|$ be the number of elements in *InfoList*. Lemma 4 provides the time complexity of RRB algorithm. We omit the proof of Lemma 4 because it is similar as that of Lemma 1.

Lemma 4. *The time complexity of the RRB algorithm is* $O(log|T_A| + |H_A| \times log(|st| + |temp| + \beta) + |InfoList|)$.

3.4 Discussion

Compared with the baseline algorithm mentioned in Section 1.3, TS, RB, and RRB algorithms improve the search performance by reducing the traversal of D_A/D_B to only once, but via different approaches. Assume that C_A represents the data object candidate set which includes all the data objects that may contribute to the optimality of some target objects, C_B denotes the target object candidate set which contains all the target objects that may have *nonzero* optimality, R_A represents the real data object set which includes all the data objects that *do* contribute to the optimality of some target objects, and R_B denotes the real target object set which contains all the target objects that *do* have *nonzero* optimality. To be more specific, $C_A = \{a \in D_A | a \in R\}, C_B = \{b \in D_B | b \notin R \wedge mindist(b, R) \leq d_c\}, R_A = \{a \in D_A | \exists b \in C_B, a \in S_b\}$, and $R_B = \{b \in C_B | S_b \neq \emptyset\}$. TS makes a conservative assumption that $C_A \approx R_A$ and $C_B \approx R_B$. Therefore, it fetches C_A (C_B) via traversing D_A (D_B) once. On the other hand, $(C_A - R_A)/(C_B - R_B)$ might be significant. RB assumes that C_A is larger than R_A and it tries to save the access of those data objects included in $(C_A - R_A)$, while RRB assumes that C_B is larger than R_B and it tries to save the access of those target objects contained in $(C_B - R_B)$. Consequently, TS favors the case where $(|C_A| - |R_A|)$ and $(|C_B| - |R_B|)$ are ignorable, RB favors the case where $(|C_A| - |R_A|) \gg (|C_B| - |R_B|)$, and RRB favors the case where $(|C_A| - |R_A|) \ll (|C_B| - |R_B|)$.

4 Performance Evaluation

In this section, we conduct extensive experiments to evaluate the efficiency and effectiveness of our proposed algorithms for answering OLS queries. All the algorithms, including Baseline, TS, RB, and RRB, were implemented in C++, and the experiments were conducted on an Intel Core 2 Duo 2.33GHz PC with 3.25GB RAM and 240GB disk, running Microsoft Windows XP Professional Edition. We first describe the experimental settings, and then, present the experimental results and our findings.

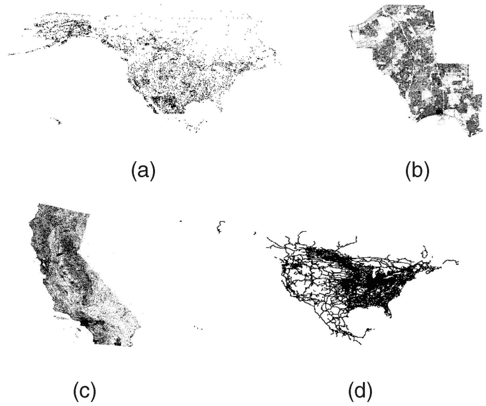

Fig. 7. Real data sets. (a) CL. (b) LB. (c) CA. (d) RS.

4.1 Experimental Setup

We have used both real and synthetic data sets in the experiments and fix each dimension of the search space to [0, 10,000]. Four real data sets are deployed with cardinality varying from 9,203 to 191,558, as summarized in Table 2. LB and CA data sets (available at http://www.census.gov/geo/www/tiger.) contain populated places and cultural landmarks, in the format of 2D point locations, in Long Beach and California, respectively. On the other hand, CL and RS data sets (available at http://www.maproom.psu.edu/dcw) represent two different layers of North America's map, with CL corresponding to cultural landmarks in the format of 2D points, and RS corresponding to railroad segments in the format of 2D segments. Although our proposed algorithms can be naturally extended to handle objects with extents, we only use point data sets in this evaluation. Consequently, we transfer RS data set into point data set by taking the middle point of each segment. Fig. 7 illustrates these four real data sets.

We have also created several synthetic data sets with dimensionality varying in the range of [2, 5] and cardinality varying between 50,000 and 450,000, following uniform and zipf distributions. The coordinates of each point in a Uniform data set are generated randomly within [0, 10,000], whereas for a Zipf data set, the coordinates follow a zipf distribution with a skew coefficient set to 0.8. In both cases, a point's coordinates on various dimensions are mutually independent.

Every data set is indexed by R*-tree [1] with 4,096 bytes page size. We investigate the performance of our proposed algorithms under various parameters, including the number k of required answer objects for an OLS query, the critical distance d_c, the size of the spatial region R, the dimensionality dim of the search space, the cardinality n of the data sets, and the buffer size bs, as presented in Table 3. In each experiment, only one parameter varies, while the others are fixed at their default values. Note that the lower left point of the spatial range R in the experiments is randomly selected from the set of data points, and all of its edges have the same length.

The I/O cost and the query cost are employed as the major performance metrics. The former is the number of page/node accesses, while the latter is the average

TABLE 3
Parameter Settings

Parameter	Setting	Default
k	$1, 4, 16, 64, 256$	16
d_c	$200, 400, 600, 800, 1000$	600
R (% of full space)	$0.25, 0.5, 1, 2, 4$	1
dimensionality dim	$2, 3, 4, 5$	2
cardinality n ($\times$1K)	$50, 150, 250, 350, 450$	$50, 250, 450$
buffer size bs (% of the tree size)	$0, 6, 12, 18, 24, 30$	0

response time of an OLS query. Here, each page access takes 10 ms, as in [39], and the query cost is the summation of the average I/O time and CPU time incurred to complete one query. Each reported value in the following diagrams is the average performance of 100 queries. Unless specifically stated, the size of LRU buffer is set to zero in the experiments.

4.2 Performance Study

As listed in Table 3, there are six parameters that might affect the performance of our algorithms. In order to evaluate the impact of each parameter, six sets of experiments are performed where we only vary one parameter in each set. Since OLS query involves two data sets (i.e., a data object set D_A and a target object set D_B), we simulate three different cases by using various data sets: 1) the data object set is *significantly larger* than the target object set, i.e., $|D_A| \gg |D_B|$ ($D_A =$ RS and $D_B =$ CL); 2) the data object set is *significantly smaller* than the target object set, i.e., $|D_A| \ll |D_B|$ ($D_A =$ CL and $D_B =$ RS); and 3) the data object set and the target object set are *about the same size*, i.e., $|D_A| \approx |D_B|$ ($D_A =$ LB and $D_B =$ CA).

Effect of k. The first set of experiments evaluates the effect of the number k of requested answer objects on the search performance, using both real and synthetic data sets. Fig. 8 depicts the query cost (in seconds) of different algorithms under various k, with the number on top of each bar representing the number of node/page accesses during the search.

As expected, baseline algorithm is worse than our proposed algorithms (i.e., TS, RB, and RRB) by several orders of magnitude. Moreover, the baseline approach is I/O-bounded in all cases, while the other algorithms are CPU-bounded, which will be confirmed by the subsequent experiments as well. This is because: 1) baseline algorithm needs to traverse the data object R-tree T_A multiple times, incurring extremely expensive I/O overhead and distance computation and 2) TS, RB, and RRB algorithms traverse the R-tree T_A/T_B *at most once*, which saves considerable node accesses. As baseline for sure performs worse than the others by factors, its performance is ignored in the rest of experimental results.

Clearly, RRB outperforms the other algorithms in all the cases. The reason behind is that RRB not only implements a single retrieval of the trees T_A and T_B, but also employs reuse technology to avoid loading the same index entries (including nodes and data points) from the disk multiple times. Although both RRB and RB enable reuse of locally available nodes/points, their performances are different. This is because of the setting of R and d_c. As R is set to 1 percent of the search space (e.g., $1{,}000 \times 1{,}000$ in a 2D space) and d_c is set to 600, the search range for potential target objects (e.g., $(600 \times 2 + 1{,}000) \times (600 \times 2 + 1{,}000) - 1{,}000 \times 1{,}000$ in a 2D space) is much larger than R. In order words, $|C_B - R_B|$ will be significantly larger than $|C_A - R_A|$ (refer to Section 3.4 for the description of C_A, C_B, R_A, and R_B). Consequently, it is more beneficial to trigger the access of target objects only when their optimal sets are very likely to be nonempty. That explains why RRB outperforms RB. If we change the setting such that the number of data objects inside R is much larger than the number of candidate target objects (i.e., $(|C_A| - |R_A|) \gg (|C_B| - |R_B|)$), RB is expected to perform better.

In addition, we observe that TS and RB perform similar in terms of I/O cost, while TS outperforms RB with respect to CPU cost. This is because when $d_c = 600$ and $R = 1\%$ of

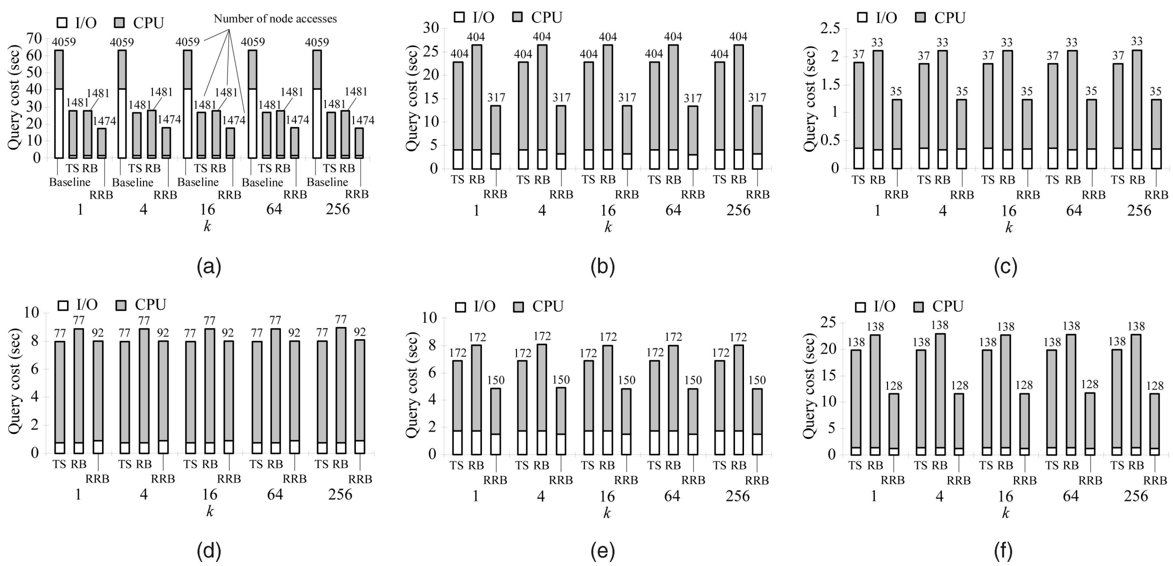

Fig. 8. Query cost versus k ($d_c = 600$, $R = 1\,\%$). (a) $D_A =$ RS and $D_B =$ CL. (b) $D_A =$ CL and $D_B =$ RS. (c) $D_A =$ LB and $D_B =$ CA. (d) $D_A = 450{,}000$ and $D_B = 50{,}000$ (Uniform). (e) $D_A = 50{,}000$ and $D_B = 450{,}000$ (Uniform). (f) $D_A = 250{,}000$ and $D_B = 250{,}000$ (Uniform).

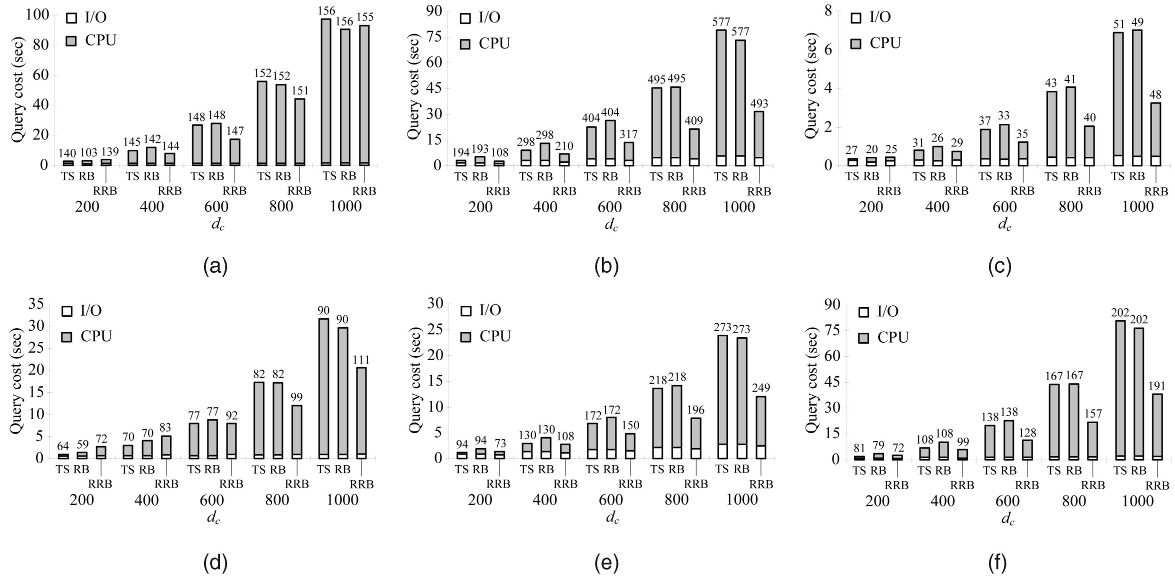

Fig. 9. Query cost versus d_c ($k = 16$, $R = 1\,\%$). (a) $D_A =$ RS and $D_B =$ CL. (b) $D_A =$ CL and $D_B =$ RS. (c) $D_A =$ LB and $D_B =$ CA. (d) $D_A = 450{,}000$ and $D_B = 50{,}000$ (Uniform). (e) $D_A = 50{,}000$ and $D_B = 450{,}000$ (Uniform). (f) $D_A = 250{,}000$ and $D_B = 250{,}000$ (Uniform).

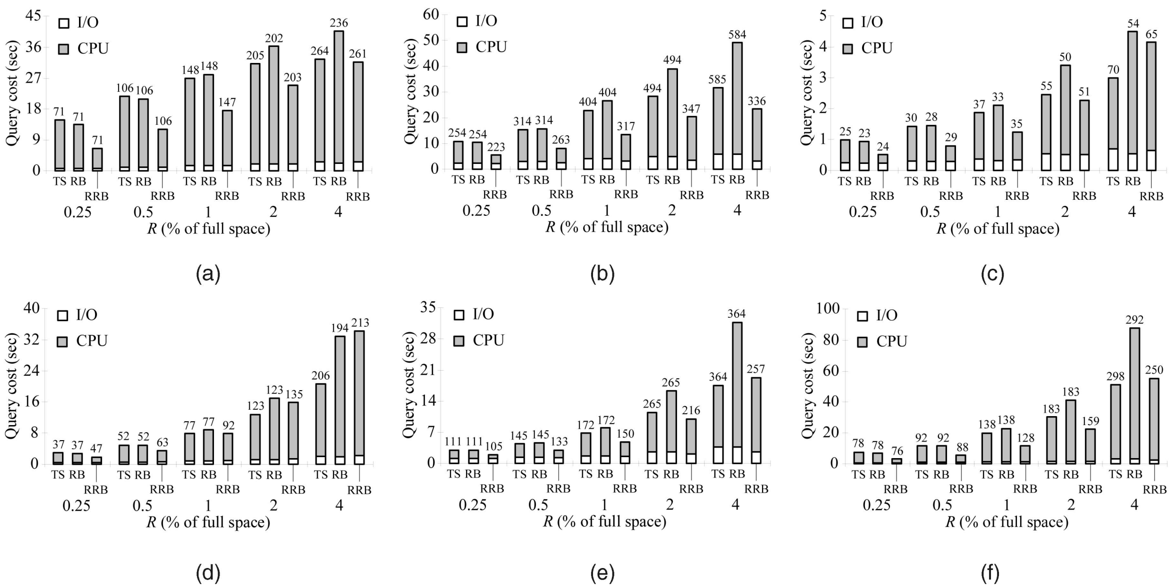

Fig. 10. Query cost versus R ($k = 16$, $d_c = 600$). (a) $D_A =$ RS and $D_B =$ CL. (b) $D_A =$ CL and $D_B =$ RS. (c) $D_A =$ LB and $D_B =$ CA. (d) $D_A = 450{,}000$ and $D_B = 50{,}000$ (Uniform). (e) $D_A = 50{,}000$ and $D_B = 450{,}000$ (Uniform). (f) $D_A = 250{,}000$ and $D_B = 250{,}000$ (Uniform).

the search space, almost all the data objects inside the range R need evaluation (i.e., $C_A \approx R_A$), and hence, RB does not save any traversal of the data objects. Furthermore, RB incurs additional overhead to reuse the entries during the query processing, as also demonstrated by the following experiments. Notice that the performance of all the methods is independent of k, since they obtain the final query result from all candidate target objects no matter how large k is.

Effect of d_c. Next, we study the impact of the critical distance d_c on the efficiency of the algorithms, by fixing $k = 16$ and varying d_c between 200 and 1,000. Fig. 9 illustrates the experimental results. As expected, the cost of all the algorithms increases with d_c, because the search space of the k-OLS query grows as d_c increases. Consistent with the performance trend observed from previous experiments, RRB performs better than the other algorithms in most of the cases. In addition, we observe that TS outperforms RB in terms of query cost when d_c is small but it loses its advantage as large d_c is encountered (e.g., $d_c = 1{,}000$). The reason behind is that RB, in order to enable reuse technique, incurs some additional overhead. Note that RB outperforms RRB when d_c is very small and $|D_A| \gg |D_B|$. This is because not all the data objects in R need evaluation, and thus, RRB which first retrieves all the data objects inside R results in some unnecessary traversal.

Effect of R. Fig. 10 plots the cost of the algorithms with respect to the spatial region R. As expected, the query cost of all the algorithms increases with the growth of R. Besides, when R is small (e.g., $R = 0.25$ percent/0.5 percent of full space), the performance of TS and RB is similar; whereas TS is obviously better than RB when R is large (e.g., $R = 2\,\%/4\%$ of full space). This is because under the current

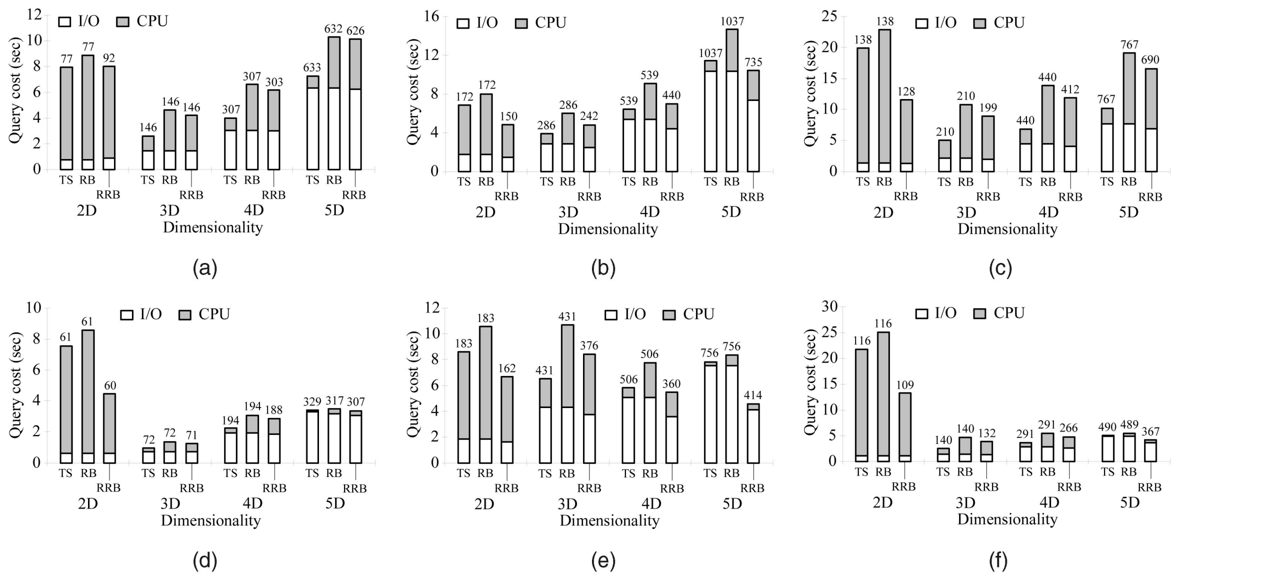

Fig. 11. Query cost versus dimensionality dim ($k = 16$, $d_c = 600$, $R = 1\%$). (a) $D_A = 450{,}000$ and $D_B = 50{,}000$ (Uniform). (b) $D_A = 50{,}000$ and $D_B = 450{,}000$ (Uniform). (c) $D_A = 250{,}000$ and $D_B = 250{,}000$ (Uniform). (d) $D_A = 450{,}000$ and $D_B = 50{,}000$ (Zipf). (e) $D_A = 50{,}000$ and $D_B = 450{,}000$ (Zipf). (f) $D_A = 250{,}000$ and $D_B = 250{,}000$ (Zipf).

setting (i.e., $d_c = 600$, and $R = 0.25/0.5/1/2/4\%$ of the search space), most, if not all, of the data objects inside R are accessed. Consequently, RB does not save many traversal of D_A, compared with TS. On the other hand, RB incurs extra overhead to scan the heap st in order to find out all the objects that contribute to the optimal set corresponding to the currently evaluated target objects. The more the target objects are evaluated, the more the overhead is. Therefore, when R becomes larger, RB suffers more from the extra overhead (under the current settings). Again, RRB performs the best and its advantage is more significant when R is small.

Effect of dimensionality dim. The next set of experiments studies the impact of the dimensionality dim on the efficiency of the algorithms. We use the synthetic data sets (containing Uniform and Zipf) and change dim from two to five, with the results depicted in Fig. 11. The I/O cost of all the algorithms increases with dim because, in general, R-trees become less efficient as the dimensionality grows [31] due to the large overlap among the MBRs. A crucial observation is that the CPU time of the algorithms in a 2D space is larger than that in higher dimensional spaces (i.e., $dim \geq 3$). This is because the object density decreases as dim increases. Thus, as the dimensionality of the search space increases, the number of data objects that contribute to the optimal set of each candidate target object decreases. Fig. 11 confirms this observation, showing that the CPU cost of the algorithms in the 2D space is the highest. In addition, as shown in the diagrams, RRB is the best approach in the 2D space, but TS outperforms the other methods in most cases when $dim \geq 3$.

Effect of cardinality n. To investigate the behaviors of the algorithms under different data set cardinalities, we use 2D Uniform and Zipf data sets. We assume that the total number of data objects and target objects remain 500,000, i.e., $|D_A| + |D_B| = 500{,}000$, and both of them follow the same distribution (either uniform or zipf). Fig. 12 measures the performance of the algorithms (in processing *16*-OLS queries) as a function of the data set cardinality n. The x-axis depicts the cardinality of the data set D_A in the unit of 1,000 points. For example, if the cardinality of D_A is 50,000 points, then the cardinality of D_B is 450,000 points. Obviously, RRB is the most efficient algorithm for all the settings. In particular, the cost of all the algorithms first increases and then drops as the relative size of D_A/D_B varies. Moreover, when the two data sets share similar cardinality, the cost of each method is the highest in the whole cost varying process.

Effect of buffer size bs. As mentioned earlier, all previous experiments are conducted without any buffer (i.e., the size of LRU buffer is set to 0). The last set of experiments examines the impact of buffer size on the performance of the proposed algorithms. We implement two types of buffers, namely, *warm up buffer* (denoted as WU) and *cold buffer* (denoted as CB). The former keeps the nodes previously accessed in the buffer to speedup the processing of queries issued later. In our experiment, we use the first 50 queries to warm up the buffer and the next 50 queries to evaluate the search performance. On the other hand, the cold buffer empties the buffer every time after a query is processed. Consequently, the processing of a new query under cold buffer always starts with an empty buffer.

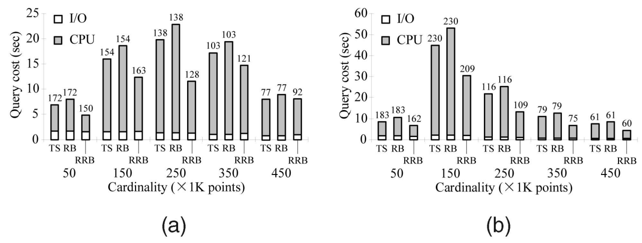

Fig. 12. Query cost versus cardinality n ($k = 16$, $d_c = 600$, $R = 1\%, dim = 2$). (a) Uniform. (b) Zipf.

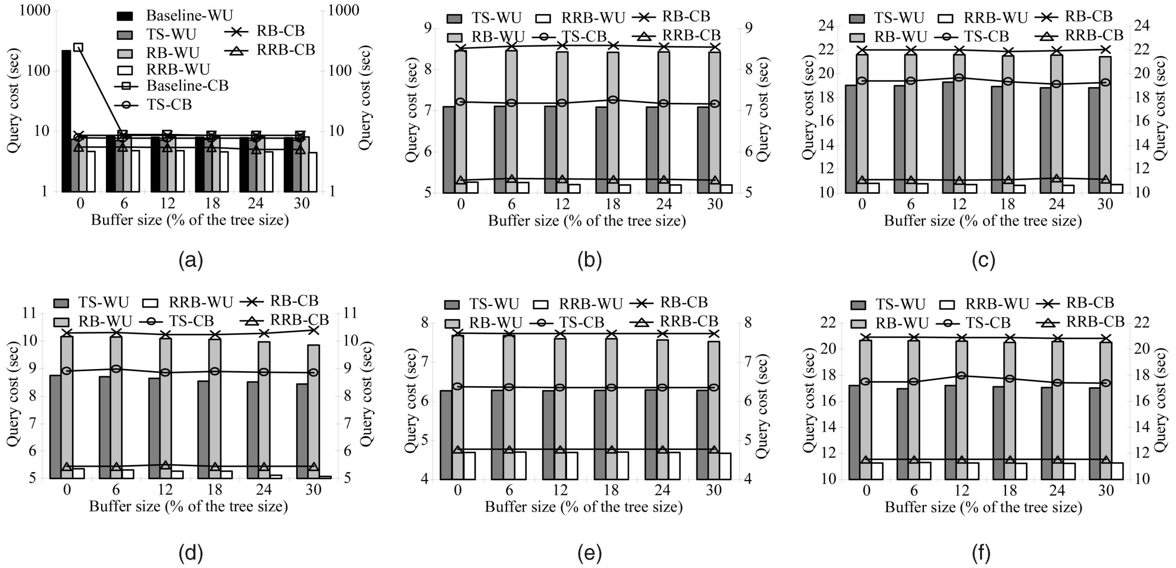

Fig. 13. Query cost versus buffer size bs ($k = 16$, $d_c = 600$, $R = 1\,\%, dim = 2\text{D}$). (a) $D_A = 450{,}000$ and $D_B = 50{,}000$ (Uniform), (b) $D_A = 50{,}000$ and $D_B = 450{,}000$ (Uniform), (c) $D_A = 250{,}000$ and $D_B = 250{,}000$ (Uniform), (d) $D_A = 450{,}000$ and $D_B = 50{,}000$ (Zipf), (e) $D_A = 50{,}000$ and $D_B = 450{,}000$ (Zipf) and (f) $D_A = 250{,}000$ and $D_B = 250{,}000$ (Zipf).

In order to save space, we integrate the results under two types of buffer schemes, using bars to represent the query cost under WU and polylines to represent the query cost under CB, as plotted in Fig. 13.

First, we analyze the performance under warm up buffer. When $bs = 0$, every node access incurs a page/node access. Hence, the cost of baseline algorithm is much higher than that of the other algorithms, because it has to traverse the R-tree T_A multiple times, as shown in Fig. 13a. When the buffer size is increased from 0 to 6 percent, the performance of baseline improves significantly since some frequently accessed nodes (e.g., *root* of T_A) are available in the buffer. However, the performance remains stable as we further increase the buffer size, which implies that only around 6 percent of T_A's nodes are accessed multiple times. On the other hand, the buffer size has a less significant impact on the performance of TS, RB, and RRB, as they only need to traverse the data set *once*. The relative performance of all the algorithms remains the same. We omit the baseline approach in the rest of the diagrams for clarity.

As for the cold buffer, the performance trends of different algorithms remain the same. In addition, we observe that for all the algorithms evaluated, the overall query cost under CB is higher than that under WU. This observation implies that different query approaches actually share certain common search paths.

5 Conclusion

In this paper, we introduce and solve a new form of spatial queries, namely, OLS search. It takes a data object set D_A, a target object set D_B, a critical distance d_c, and a spatial region R as inputs, and returns those target objects *outside* R with maximal *optimality*. We develop the optimality metric, formalize the OLS query and its generalization (i.e., k-OLS query), and propose three algorithms, including TS, RB, and RRB, for efficient k-OLS query processing. Finally, a comprehensive empirical study using both real and synthetic data sets has been conducted to verify the performance of our proposed algorithms in terms of efficiency and effectiveness.

The work reported in this paper presents our first step with respect to OLS queries. There are certain limitations. First, in the current work, we only consider an euclidian space and we plan to extend our methods to other distance metrics (e.g., network distance in road networks). Second, the current algorithms are designed to mainly reduce the I/O cost via traversing the data object set and target object set only once. However, it is also important to improve the CPU time. Hence, our next step is to design the algorithms that can improve both CPU time and I/O cost. Third, the current algorithms are k-insensitive as they have to evaluate all the candidate target objects with nonzero optimality no matter how large the k is. We are currently working on the incremental algorithms which can terminate the evaluation once the k target objects with maximal optimality are identified. In addition, we are also interested in developing a cost model to estimate the execution time of different OLS query processing algorithms, which can facilitate query optimization and reveal new problem characteristics that could lead to even faster algorithms.

Acknowledgments

This research is partly supported by the Office of Research, Singapore Management University.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles,” *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 322-331, 1990.

[2] S. Berchtold, D.A. Keim, and H.-P. Kriegel, “The X-Tree: An Index Structure for High-Dimensional Data,” *Proc. Very Large Data Bases Conf. (VLDB '96)* , pp. 28-39, 1996.

[3] C. Bohm and F. Krebs, "High Performance Data Mining Using the Nearest Neighbor Join," *Proc. Int'l Conf. Data Mining (ICDM '02)*, pp. 43-50, 2002.

[4] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins Using R-Trees," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 237-246, 1993.

[5] K.L. Cheung and A.W.-C. Fu, "Enhanced Nearest Neighbour Search on the R-Tree," *SIGMOD Record*, vol. 27, no. 3, pp. 16-21, 1998.

[6] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest Pair Queries in Spatial Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 189-200, 2000.

[7] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Algorithms for Processing *k*-Closest-Pair Queries in Spatial Databases," *Data & Knowledge Eng.*, vol. 49, no. 1, pp. 67-104, 2004.

[8] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Multi-Way Distance Join Queries in Spatial Databases," *GeoInformatica*, vol. 8, no. 4, pp. 373-402, 2004.

[9] K. Deng, X. Zhou, H. Shen, K. Xu, and X. Lin, "Surface *k*-NN Query Processing," *Proc. Int'l Conf. Data Eng. (ICDE '06)*, p. 78, 2006.

[10] Y. Du, D. Zhang, and T. Xia, "The Optimal-Location Query," *Proc. Int'l Symp. Spatial and Temporal Databases (SSTD '05)*, pp. 163-180, 2005.

[11] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. Abbadi, "Constrained Nearest Neighbor Queries," *Proc. Int'l Symp. Spatial and Temporal Databases (SSTD '01)*, pp. 257-278, 2001.

[12] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, "Algorithms for Nearest Neighbor Search on Moving Object Trajectories," *GeoInformatica*, vol. 11, no. 2, pp. 159-193, 2007.

[13] Y. Gao, C. Li, G. Chen, L. Chen, X. Jiang, and C. Chen, "Efficient *k*-Nearest-Neighbor Search Algorithms for Historical Moving Object Trajectories," *J. Computer Science and Technology*, vol. 22, no. 2, pp. 232-244, 2007.

[14] Y. Gao, C. Li, G. Chen, Q. Li, and C. Chen, "Efficient Algorithms for Historical Continuous *k*nn Query Processing over Moving Object Trajectories," *Proc. Joint Int'l Conf. Asia-Pacific Web Conf. (APWeb)/Web-Age Information Management (WAIM '07)*, pp. 188-199, 2007.

[15] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 47-57, 1984.

[16] G.R. Hjaltason and H. Samet, "Incremental Distance Join Algorithms for Spatial Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 237-248, 1998.

[17] G.R. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *ACM Trans. Database Systems*, vol. 24, no. 2, pp. 265-318, 1999.

[18] H. Hu and D.L. Lee, "Range Nearest-Neighbor Query," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, no. 1, pp. 78-91, Jan. 2006.

[19] F. Korn and S. Muthukrishnan, "Influence Sets Based on Reverse Nearest Neighbor Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 201-212, 2000.

[20] N. Koudas and K. Sevcik, "High Dimensional Similarity Joins: Algorithms and Performance Evaluation," *IEEE Trans. Knowledge and Data Eng.*, vol. 12, no. 1, pp. 3-18, Jan./Feb. 2000.

[21] N. Mamoulis and D. Papadias, "Multiway Spatial Joins," *ACM Trans. Database Systems*, vol. 26, no. 4, pp. 424-475, 2001.

[22] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias, "Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitorin," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 634-645, 2005.

[23] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao, "A Threshold-Based Algorithm for Continuous Monitoring of *k* Nearest Neighbors," *IEEE Trans. Knowledge and Data Eng.*, vol. 17, no. 11, pp. 1451-1464, Nov. 2005.

[24] K. Mouratidis, M. Yiu, D. Papadias, and N. Mamoulis, "Continuous Nearest Neighbor Monitoring in Road Networks," *Proc. Very Large Data Bases Conf. (VLDB '06)*, pp. 43-54, 2006.

[25] A. Nanopoulos, Y. Theodoridis, and Y. Manolopoulos, "C2P: Clustering Based on Closest Pairs," *Proc. Very Large Data Bases Conf. (VLDB '01)* , pp. 331-340, 2001.

[26] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer, "Towards an Analysis of Range Query Performance in Spatial Data Structures," *Proc. ACM Symp. Principles of Database Systems (PODS '93)*, pp. 214-221, 1993.

[27] D. Papadias and D. Arkoumanis, "Approximate Processing of Multiway Spatial Joins in Very Large Databases," *Proc. Int'l Conf. Extending Database Technology (EDBT '02)*, pp. 179-196, 2002.

[28] D. Papadias, N. Mamoulis, and D. Theodoridis, "Processing and Optimization of Multiway Spatial Joins Using R-Trees," *Proc. ACM Symp. Principles of Database Systems (PODS '99)*, pp. 44-55, 1999.

[29] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis, "Group Nearest Neighbor Queries," *Proc. Int'l Conf. Data Eng. (ICDE '04)*, pp. 301-312, 2004.

[30] D. Papadias, Y. Tao, K. Mouratidis, and K. Hui, "Aggregate Nearest Neighbor Queries in Spatial Databases," *ACM Trans. Database Systems*, vol. 30, no. 2, pp. 529-576, 2005.

[31] A. Papadopoulos and Y. Manolopoulos, "Performance of Nearest Neighbor Queries in R-Trees," *Proc. Int'l Conf. Database Theory (ICDT '97)*, pp. 394-408, 1997.

[32] A. Papadopoulos, A. Nanopoulos, and Y. Manolopoulos, "Processing Distance Join Queries with Constraints," *The Computer J.*, vol. 49, no. 3, pp. 281-296, 2006.

[33] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbor Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 71-79, 1995.

[34] J. Shan, D. Zhang, and B. Salzberg, "On Spatial-Range Closest-Pair Query," *Proc. Int'l Symp. Spatial and Temporal Databases (SSTD '03)*, pp. 252-269, 2003.

[35] H. Shin, B. Moon, and S. Lee, "Adaptive Multi-Stage Distance Join Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 343-354, 2000.

[36] H. Shin, B. Moon, and S. Lee, "Adaptive and Incremental Processing for Distance Join Queries," *IEEE Trans. Knowledge and Data Eng.*, vol. 15, no. 6, pp. 1561-1578, Nov./Dec. 2003.

[37] Y. Shou, N. Mamoulis, H. Cao, D. Papadias, and D.W. Cheung, "Evaluation of Iceberg Distance Joins," *Proc. Int'l Symp. Spatial and Temporal Databases (SSTD '03)*, pp. 270-288, 2003.

[38] Z. Song and N. Roussopoulos, "*k*-Nearest Neighbor Search for Moving Query Point," *Proc. Int'l Symp. Spatial and Temporal Databases (SSTD '01)*, pp. 79-96, 2001.

[39] Y. Tao, D. Papadias, X. Lian, and X. Xiao, "Multidimensional Reverse *k*NN Search," *The Very Large Data Bases J.*, vol. 16, no. 3, pp. 293-316, 2007.

[40] Y. Tao, D. Papadias, and Q. Shen, "Continuous Nearest Neighbor Search," *Proc. Int'l Conf. Very Large Data Bases (VLDB '02)*, pp. 287-298, 2002.

[41] X. Xiong, M. Mokbel, and W. Aref, "SEA-CNN: Scalable Processing of Continuous *k*-Nearest Neighbor Queries in Spatio-Temporal Databases," *Proc. Int'l Conf. Data Eng. (ICDE '05)*, pp. 643-654, 2005.

[42] X. Yu, K. Pu, and N. Koudas, "Monitoring *k*-Nearest Neighbor Queries over Moving Objects," *Proc. Int'l Conf. Data Eng. (ICDE '05)*, pp. 631-642, 2005.

[43] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao, "All-Nearest-Neighbors Queries in Spatial Databases," *Proc. Int'l Conf. Scientific and Statistical Database Management (SSDBM '04)*, pp. 297-306, 2004.

[44] M. Zhu, D. Papadias, J. Zhang, and D.L. Lee, "Top-*k* Spatial Joins," *IEEE Trans. Knowledge and Data Eng.*, vol. 17, no. 4, pp. 567-579, Apr. 2005.

Yunjun Gao received the master's degree in computer science from Yunnan University, China, in 2005, and the PhD degree in computer science from Zhejiang University, China, in 2008. He is currently a postdoctoral research fellow in the School of Information Systems, Singapore Management University, Singapore. His research interests include spatial databases, spatiotemporal databases, mobile/pervasive computing, and geographic information systems. He is a member of the IEEE, the ACM, and the ACM SIGMOD.

Baihua Zheng received the bachelor's degree in computer science from Zhejiang University, China, in 1999, and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong, in 2003. She is currently an assistant professor in the School of Information Systems, Singapore Management University, Singapore. Her research interests include mobile/pervasive computing and spatial databases. She is a member of the IEEE and the ACM.

Gencai Chen is a professor in the College of Computer Science, Zhejiang University, China. He was a visiting scholar in the Department of Computer Science, State University of New York at Buffalo, from 1987 to 1988, and the winner of the special allowance, conferred by the State Council of China in 1997. He is currently a vice dean of the College of Computer Science, the director of the Computer Application Engineering Center, and the vice director of the Software Research Institute, Zhejiang University. His research interests include database systems, artificial intelligence, and CSCW.

Qing Li is a professor in the Department of Computer Science, City University of Hong Kong, where he joined as a faculty member in September 1998. Before that, he taught at the Hong Kong Polytechnic University, The Hong Kong University of Science and Technology, and The Australian National University (Canberra, Australia). He is a guest professor at the University of Science and Technology of China, a visiting professor at the Institute of Computing Technology (Knowledge Grid), Chinese Academy of Science (Beijing, China), an adjunct professor at the Hunan University (Changsha, China), and a guest professor (software technology) at Zhejiang University (Hangzhou, China). His research interests include object modeling, multimedia databases, and Web services. He is a senior member of the IEEE and a member of the ACM SIGMOD and the IEEE Technical Committee on Data Engineering. He is the chairperson of the Hong Kong Web Society and also served/is serving as an executive committee (EXCO) member of the IEEE-Hong Kong Computer Chapter and the ACM Hong Kong Chapter. In addition, he serves as a councilor of the Database Society of Chinese Computer Federation, a councilor of the Computer Animation and Digital Entertainment Chapter of Chinese Computer Imaging and Graphics Society, and is a steering committee member of DASFAA, ICWL, and the International WISE Society.