Information Sciences 298 (2015) 98–117

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

On efficient *k*-optimal-location-selection query processing in metric spaces

Yunjun Gao [a,*], Shuyao Qi [b], Lu Chen [a], Baihua Zheng [c], Xinhan Li [a]

[a] College of Computer Science, Zhejiang University, Hangzhou, China
[b] Department of Computer Science, The University of Hong Kong, Hong Kong, China
[c] School of Information Systems, Singapore Management University, Singapore

ARTICLE INFO

Article history:
Received 28 November 2013
Received in revised form 24 November 2014
Accepted 26 November 2014
Available online 3 December 2014

Keywords:
Optimal location selection
k-optimal-location-selection query
Metric spaces
Query processing
Spatial database

ABSTRACT

This paper studies the problem of *k-optimal-location-selection* (*k*OLS) *retrieval* in metric spaces. Given a set D_A of customers, a set D_B of locations, a constrained region R, and a critical distance d_c, a *metric* *k*OLS (M*k*OLS) query retrieves k locations in D_B that are *outside* R but have the *maximal optimality scores*. Here, the optimality score of a location $l \in D_B$ located outside R is defined as the number of the customers in D_A that are inside R and meanwhile have their distances to l bounded by d_c according to a certain similarity metric (e.g., L_1-norm, L_2-norm, etc.). The existing *k*OLS methods are not sufficient because they are applicable only to the Euclidean space, and are *not sensitive* to k. In this paper, for the first time, we present an efficient algorithm for *k*OLS query processing in metric spaces. Our solution employs metric index structures (i.e., M-trees) on the datasets, enables several *pruning rules*, utilizes the advantages of *reuse technique* and *optimality score estimation*, to support a wide range of data types and similarity metrics. In addition, we extend our techniques to tackle two interesting and useful variants, namely, M*k*OLS queries with *multiple* or *no* constrained regions. Extensive experimental evaluation using both real and synthetic data sets demonstrates the effectiveness of the presented pruning rules and the performance of the proposed algorithms.

1. Introduction

Given an object set D_A, a location set D_B, a constrained region R, and a distance parameter d_c, a *k-optimal-location-selection* (*k*OLS) query returns top-k optimal locations in D_B that are located outside R. Note that, in this paper, the optimality score of a location $l \in D_B$ is defined as the number of the objects in D_A that are within R and have their distances to l bounded by d_c. *k*OLS queries are useful in a large number of applications such as decision making and resource/service planning. Three potential applications are listed below as three examples.

* Corresponding author at: College of Computer Science, Zhejiang University, 38 Zheda Road, Hangzhou 310027, China. Tel.: +86 571 8765 1613; fax: +86 571 8795 1250.
E-mail addresses: gaoyj@zju.edu.cn (Y. Gao), syqi2@cs.hku.hk (S. Qi), luchen@zju.edu.cn (L. Chen), bhzheng@smu.edu.sg (B. Zheng), xhli@zju.edu.cn (X. Li).

http://dx.doi.org/10.1016/j.ins.2014.11.038
0020-0255/

Application 1 (*Pizza restaurant location selection*). Consider that, as shown in Fig. 1, Pizza Hut would like to open a new store nearby a residential area R. The main customer base of this new store is the residents D_A in R. Because of the 30-min delivery guarantee, the store p may only provide the pizza delivery service to the residents with their distances to p bounded by a certain distance d_c (e.g., 2 km). In addition, due to certain restriction conditions (e.g., imagine R is a campus which does not open to any public restaurant), the new Pizza Hut restaurant has to be located outside R. If the potential customer base for the delivery service is the most important, given a set D_B of potential locations, the kOLS query can help the decision-maker to identify an appropriate pizza restaurant location that covers the largest number of residents (e.g., l_2 in Fig. 1).

Application 2 (*Data center planning*). A data center is a facility used to house computer systems and associated components. For security reason, the data servers D_A are usually placed in a safe region R. There are a set D_B of locations outside R such that we can install switches and routers, which can transport traffic between the servers and the outside world. The data is forwarded from a source to a destination via router(s) one by one. In general, the fewer the routers the data pass, the safer and the faster a transfer would be. In order to ensure a fast and secure service, we assume that a router r can reach those servers that are within d_c hops from r. Under this assumption, kOLS search can be employed to analyze the optimality of different locations for routers, and hopefully we need less routers to connect to more servers.

Application 3 (*New computer promotion*). Consider a computer manufacturer is planning to launch new computers to attract more customers. Every computer can be modeled as a d-dimensional vector, to represent its price, size, color and other performance features. Similarly, a corresponding d-dimensional vector can also be used to denote a customer's preference. If the similarity (e.g., L_∞-norm) between a computer and a customer's preference is no larger than d_c, the customer can be considered as a potential customer who may buy the computer. Based on this assumption, a kOLS query could be helpful to select optimal computers from a computer dataset D_B to attract more customers in D_A. In this case, the constrained region R can be considered as a "circle" to select closely related computers, with a popular computer as its center and a predefined/customer-specified distance as its radius.

Although the concept of kOLS search is not new [16], the existing work only considers the *Euclidean space*, where the distance between two objects is measured by the *Euclidean distance*. In reality, not all the applications can be fit into the Euclidean space, and the Euclidean distance might not be able to capture the distance between objects. In Application 1, the distance from a Pizza Hut restaurant location to a resident address cannot be captured by the Euclidean distance but the road network distance; In Application 2, the distance from a data server to a router is measured by the number of hops but not the Euclidean distance. Moreover, Application 3 generalizes the problem and goes beyond typical distance and location concepts. Specifically, each computer (customer's preference) is represented as a d-dimensional feature vector, and the distance (e.g., L_∞-norm) between a computer and a customer's preference is defined according to the feature vectors. In a word, there are many applications in real life that require a more general solution for flexible distance measurements and/or object representations. Motivated by this, in this paper, we propose a *metric k-optimal-location-selection* (MkOLS) *query* to support the kOLS query in a metric space. Due to the difference between the Euclidean space and the metric space, the original solution for the Euclidean space presented in [16] cannot be applied to answer MkOLS retrieval efficiently, and new efficient algorithms are needed. To the best of our knowledge, there is no prior work on this problem.

A naive solution to tackle MkOLS search is to, for each location $l \in D_B$, we can quantify its optimality score by traversing the object set D_A; and then, we return those k locations with the maximum optimality scores. This approach is straightforward, whereas it is *very inefficient* due to the following two deficiencies. First, it needs to traverse D_A *multiple times*, resulting in *high* I/O and CPU costs. Second, it is *insensitive* to k and hence has to scan all the objects in D_B even when $k \ll |D_B|$.

In this paper, we propose an efficient algorithm for MkOLS query processing, assuming that both D_A and D_B are indexed by M-trees [9]. However, the presented methodology is not limited to the M-tree only, and it can also be applied to other metric indexes [17]. In particular, our solution enables several *pruning rules*, utilizes the advantages of *reuse technique* and *optimality*

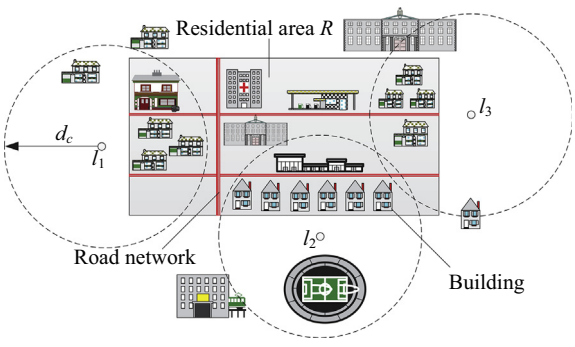


Fig. 1. Illustration of pizza restaurant location selection.

score estimation, requires *no* detailed representations of objects, and can be applied as long as their mutual distances can be computed and the distance metric satisfies the triangle inequality. Furthermore, our techniques can be easily extended to solve some interesting and useful variants of M*k*OLS queries. In brief, the key contributions of this paper are summarized as follows:

- We formalize the M*k*OLS query, a *new* and valuable addition to the family of optimal location selection problems and queries in metric spaces.
- We develop a suite of pruning rules to significantly reduce query costs and present an efficient algorithm for processing *exact* M*k*OLS retrieval using M-trees.
- We extend our techniques to handle two interesting and useful variants of M*k*OLS queries, namely, M*k*OLS search with *multiple* or *no* constrained regions.
- We conduct extensive experiments to verify the effectiveness of our developed pruning rules and the performance of our proposed algorithms.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 formulates the M*k*OLS query, presents pruning rules, and describes the Baseline algorithm. Section 4 elaborates an efficient M*k*OLS search algorithm. Section 5 extends our techniques to address two M*k*OLS query variants. Considerable experimental results and our findings are reported in Section 6. Finally, Section 7 concludes the paper with some directions for future work.

2. Related work

In this section, we review the previous work related to M*k*OLS retrieval, including facility location problems and query processing in metric spaces.

2.1. Facility location problems

Given an object set and a location set, the *facility location* (FL) problem is to retrieve the optimal location(s) that can attract the most objects. The existing work can be classified into two categories, i.e., *max-inf* problems and *min-dist* problems. Our problem studied in this paper belongs to the first category, and it is a significant addition to the family of FL problems.

2.1.1. Max-inf problems

Max-inf problems aim at finding the locations with the maximum *influence*, and there are two different ways to measure the influence. The first one is based on the concept of *reverse nearest neighbor* (RNN) query [20], and it quantifies the influence of an object o as the number of the objects taking o as their nearest neighbors. Du et al. [13] propose the algorithms for selecting a location in a specified spatial region, where the influence of a location is defined as the total weight of its RNNs in an object set. Cabello et al. [5] introduce a facility location problem using the MAXCOV optimization criterion, which finds the regions in a data space to maximize the numbers of RNNs for the objects in these regions. Huang et al. [18] develop two branch and bound algorithms for finding the top-k most influential locations in a given set of locations. Xia et al. [34] address the problem of finding the top-t most influential sites inside a specified spatial region. Zhang et al. [37] study the problem of finding the optimal location that has maximum aggregate weight on multiple types of objects. Note that, all these efforts differ from ours in *optimality functions* and some *settings* (e.g., our work aims at *metric spaces* instead of *Euclidean spaces*), and thus are not applicable to M*k*OLS search. On the other hand, the second one is to measure the optimality of a location beyond RNN. Gao et al. [16] first identify the *k*OLS query and propose three algorithms to tackle it. Xiao et al. [35] explore optimal location (OL) queries in road networks. In particular, they develop a unified framework to solve three interesting variants of OL queries. It is worth mentioning that, however, these solutions are specifically designed for the Euclidean space or the road network, and hence cannot be used to handle M*k*OLS retrieval.

2.1.2. Min-dist problems

Min-dist problems aim at minimizing the average distance between the facilities and their corresponding customers. Zhang et al. [36] discuss the min-dist optimal-location problem where, given a set C, a facility set F, and a spatial region Q, the goal is to find the locations in Q such that, if a new facility is built at any one of these locations, the average distance from each customer to its closest facility is minimized. Mouratidis et al. [23] study the k-medoid query, which returns a set P' of k medoids from a point set P that minimizes the average distance between every point in P and its nearest medoid in P'. Qi et al. [25] investigate the min-dist location selection problem which, given a customer set C, a facility set F, and a location set L, finds a location in L for establishing a new facility such that the average distance from a customer to its nearest facility is minimized. Recently, Chen et al. [8] explore the problem to find an optimal location that minimizes the maximum distance to client objects, based on road network graphs. It is worth noting that, all the above works are different from ours in that (i) they try to minimize the average distance, whereas we aim to maximize the number of objects and (ii) none of these approaches is designed for metric spaces. Thus, they are not applicable to answer M*k*OLS search.

More recently, Didandeh et al. [11] introduce the facility location problem to locate a set of facilities with respect to a dynamic set of demands. Wang et al. [33] study the target set selection problem on social networks. Fort and Sellarès [14] explore the k-influence region problem using GPU parallel approach. Rahmani and Mirhassani [26] investigate the capacitated facility location problem (CFLP), which aims to determine how to locate facilities and move commodities, such that the customers' demands are satisfied and the total cost is minimized. Zhang and Chow [38] exploit personalized influence to facilitate location recommendations using the geographical and social influences. However, all these works are clearly different from ours, since (i) they take other factors (e.g., GPU, social network, etc.) into consideration to address facility location problems and (ii) none of their solutions is designed for the metric space. Thus, they are not applicable to solve our studied problem.

2.2. Querying metric spaces

Since indexes can accelerate query processing, following the most approaches in the relevant literature [1,2,7,29], we assume, in this paper, that each dataset is indexed by an M-tree [9]. In an M-tree, an intermediate entry e records (i) a routing object $e.o$ that is a selected object in the subtree sub_e rooted by e, (ii) a covering radius $e.r$ which is set to the maximum distance between e and the objects in sub_e, and (iii) a parent distance $e.pdist$ corresponding to the distance from e to the routing object of the parent entry e_p referencing the node containing e. All the objects in sub_e lie in e's cluster sphere centered at $e.o$ with radius $e.r$. A leaf entry o stores the details of an object and the distance $o.pdist$ to its parent entry. As to be used later, the *minimum distance* and *maximum distance* between an entry e and an object o is the smallest and largest distances from o to any object in sub_e, respectively, i.e.,

$$mindist(e,o) = MAX(dist(e.o,o) - e.r, 0)$$
$$maxdist(e,o) = dist(e.o,o) + e.r$$

Range and k-nearest neighbor (kNN) queries in metric spaces have been well-studied in the database literature [6]. Brin [4] presents an algorithm based on Geometric NN Access Tree (GNAT). Ciaccia et al. [9] adopt a branch-and-bound technique using M-tree for processing range and kNN queries in metric spaces. Clarkson [10] proposes two data structures, i.e. D(S) and M(S, Q), for NN retrieval in metric spaces. Skopal et al. [28] introduce the PM-tree, a variation of M-tree that combines M-tree with the pivot-based methods for similarity search. Tellez et al. [30] propose an NN algorithm based on a new metric indexing technique with an algorithmic mechanism to lift the performance of otherwise rigid metric indexes. Ares et al. [3] present the *cluster reduction* approach for similarity search in metric spaces. Recently, Doulkeridis et al. [12] address P2P similarity search in metric spaces, where data are horizontally distributed across a P2P network. Vlachou et al. [32] propose a framework for distributed similarity search, in which each participating peer preserves its own data autonomously and is indexed by an M-tree.

Achtert et al. [1] propose an approach for efficient reverse k-nearest neighbor (RkNN) query in arbitrary metric spaces, which uses conservative and progressive distance approximations to filter out true drops and true hits. Tao et al. [29] present a two-stage algorithm for RkNN search and develop several novel pruning heuristics to boost search performance. Achtert et al. [2] give a solution to RkNN retrieval in a dynamic environment, which integrates the potentials of self-pruning and mutual pruning to achieve optimal pruning power and reduce the query time accordingly. Liu et al. [22] present an efficient algorithm for reverse furthest neighbors (RFN) query.

Jacox and Samet [19] introduce a *Quickjoin* algorithm for similarity join, which recursively partitions the objects until each partition contains a few objects, where a nested-loop join is employed. Paredes and Reyes [24] develop a new metric index, i.e., *coined List of Twin Clusters (LTC)*, for answering similarity joins. Kurasawa et al. [21] propose a new divide-and-conquer-based k-closest pair query method, i.e., *Adaptive Multi-Partitioning (AMP)*, in metric spaces. Silva and Pearson [27] develop *DBSimJoin*, a physical similarity join database operator for datasets in any metric space.

Chen and Lian [7] and Fuhry et al. [15] study skyline query in metric spaces. More Recently, Tiakas et al. [31] investigate metric based top-k dominating queries, which combines top-k and skyline queries under generic metric spaces. Although query processing in metric spaces has been well studied, we would like to highlight that there is *no* prior work on answering top-k optimal location selection queries in metric spaces. To the best of our knowledge, the work presented in this paper is the first attempt.

3. Preliminaries

In this section, we formally define the MkOLS query, propose several effective pruning rules to speed up the search, and present a *baseline algorithm* for answering MkOLS search. Table 1 lists the symbols used frequently in the rest of this paper. The example depicted in Fig. 2 serves as a running example in the rest of the paper.

3.1. Problem formulation

We first formalize the concepts of *optimal set* and *optimality score* in Definitions 1 and 2, respectively, based on which MkOLS retrieval is defined in Definition 3.

Table 1
Symbols and description.

Notation	Description
D_A	A set of objects in metric spaces
D_B	A set of locations in metric spaces
T_A	The M-tree/COM-tree on D_A
T_B	The M-tree on D_B
k	The number of required location(s)
R	A region centered at $R.o$ with radius $R.r$
d_c	A critical distance
S_b	The optimal set of a location b in D_B
$\|\|b,\ S_b\|\|$	The accumulated distance of a location b in D_B
$b.OPT$	The optimality score of a location b in D_B
$B.EST$	The estimated optimality score of a (leaf or non-leaf) entry $B \in T_B$

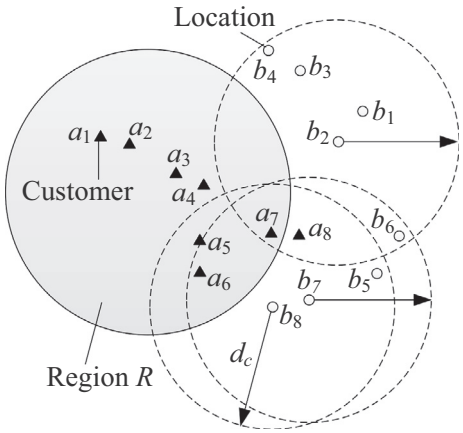


Fig. 2. A running example for M*k*OLS search.

Definition 1 (*Optimal set*). Given an object set D_A, a location set D_B, a region R, and a distance d_c, the optimal set S_{b_j} of a location $b_j \in D_B$ located outside R is formed by all the objects $a_i \in D_A$ that are within R and meanwhile have their distances to b_j bounded by d_c, i.e., $S_{b_j} = \{a_i | a_i \in D_A \wedge a_i \in R \wedge dist(a_i, b_j) \leqslant d_c\}$, where $dist(a_i, b_j)$ refers to a metric distance between a_i and b_j that could be of any type of objects in a metric space.

M*k*OLS search aims to find top-k locations with the largest optimal sets. However, there might be *multiple* locations sharing the same size of optimal sets, i.e., there is a tie. As shown in Fig. 2, for a given distance d_c, b_7 and b_8 have the same optimal sets, i.e., $S_{b_7} = S_{b_8} = \{a_5, a_6, a_7, a_8\}$. Consequently, we need to define a tie breaker. Although different applications may prefer various tie breakers, we employ the *accumulated distance*. For a specified location $b_j \in D_B$, its accumulated distance, denoted by $||S_{b_j}, b_j||$, is defined as $\sum_{a_i \in S_{b_j}} dist(a_i, b_j)$, in which $dist(x, y)$ denotes a certain metric distance function defined by the application. As an example, in the context of Application 1, $dist(x, y)$ can be defined as the network distance from x to y, which reflects the travel distance from the Pizza Hut restaurant to a residential address. If two locations share the same size of potential customer base (i.e., the optimal set), we prefer the one closer to the potential customers, i.e., having a smaller accumulated distance. Continuing the above example, location b_8 is better than location b_7 due to the smaller accumulated distance. Formally, the *optimality score* of a location is formulated below, by considering both optimal set size and the accumulated distance.

Definition 2 (*Optimality score*). Given an object set D_A, a location set D_B, a region R, and a distance $d_c, \forall b_j \in D_B \wedge b_j \notin R$, its optimality score is defined as

$$b_j.OPT = |S_{b_j}| - \frac{||S_{b_j}, b_j||}{d_c \times |S_{b_j}| + 1}$$

Note that, $|S_{b_j}|$ represents the cardinality of S_{b_j}, and $\frac{||S_{b_j}, b_j||}{d_c \times |S_{b_j}| + 1}$ is within the range of $[0, 1)$. In other words, the Definition 2 considers the size of the optimal set of a location b_j as the major factor when evaluating b_j's optimality score. The accumulated distance plays a role only if two locations share the same size of the optimal set. Formally, $\forall b_i, b_j \in D_B \wedge b_i, b_j \notin R, |S_{b_i}| >$

$|S_{b_j}| \rightarrow b_i.OPT > b_j.OPT$; and $|S_{b_i}| = |S_{b_j}| \wedge ||S_{b_i}, b_i|| < ||S_{b_j}, b_j|| \rightarrow b_i.OPT > b_j.OPT$. Based on the concepts of the optimal set and the optimality score, the MkOLS query is defined as follows. Notice that, MkOLS retrieval might return less than k locations.

Definition 3 (*MkOLS search*). Given an object set D_A, a location set D_B, a region R, a distance d_c, and an integer $k(\geqslant 1)$ in a metric space, a *metric k-optimal-location-selection (MkOLS) query* finds k locations in D_B having the maximal optimality score among all the locations located outside R, i.e., MkOLS $(D_A, D_B, R, d_c, k) = \{Res | Res \subseteq D_B, |Res| \leqslant k,$ and $\forall b_i \in Res, \forall b_j \in (D_B - Res) \wedge b_j \notin R, b_i.OPT \geqslant b_j.OPT$ and $b_i.OPT > 0\}$.

3.2. Pruning rules

Due to the lack of geometric properties, query processing in metric spaces is naturally more challenging than that in Euclidean spaces. In the following, we propose several pruning rules that will be used in our search algorithms. Note that, all these pruning rules are based on the two assumptions: (i) the query is purely processed based on mutual distances but nothing else and (ii) two M-trees T_A and T_B are built over D_A and D_B, respectively. Next, we first present the rules to prune the entries in T_A, and then present the rules to discard the entries in T_B.

Rule 1. For an entry $A \in T_A, A$ can be safely pruned away if $mindist(A, R) > 0$, in which $mindist(A, R)$ represents the minimum distance between A and a given constrained region R.

Proof. If $mindist(A, R) > 0, A$ is outside R. In other words, A does not contain any object located within R and thus can be pruned. □

Rule 2. Given an entry $A \in T_A$ and its parent entry A_p, if $dist(A_p.o, R.o) - A.pdist > A.r + R.r, A$ can be discarded safely.

Proof. Given three objects $A.o, A_p.o$, and $R.o$ in a metric space, $dist(A_p.o, R.o) \leqslant dist(A.o, R.o) + dist(A_p.o, A.o) = dist(A.o, R.o) + A.pdist$, due to the triangle inequality. If $dist(A_p.o, R.o) - A.pdist > A.r + R.r$, then $dist(A.o, R.o) \geqslant dist(A_p.o, R.o) - A.pdist > A.r + R.r$, i.e., $mindist(A, R) = dist(A.o, R.o) - A.r - R.r > 0$. Thus, the entry A can be safely pruned according to Rule 1, and the proof completes. □

Both Rules 1 and 2 are employed to prune away the entries $A \in T_A$ based on $mindist(A, R)$. Rule 1 directly checks $mindist(A, R)$, while Rule 2 derives the value of $mindist(A, R)$ with the help of A's parent entry A_p. Consider, for instance, in Fig. 3, the leaf entry a_1 can be discarded as $mindist(a_1, R) > 0$, and the intermediate entry A_1 can also be safely pruned as $mindist(A_1, R) > 0$. On the other hand, the entry A_2 containing objects a_4 and a_5 is within R as $mindist(A_2, R) = 0$. Consequently, A_2 cannot be pruned away.

Rule 3. For an entry $B \in T_B$, if $maxdist(B, R.o) - R.r \leqslant 0$ or $mindist(B, R) > d_c, B$ can be safely pruned.

Proof. If $maxdist(B, R.o) - R.r \leqslant 0$, then B is completely within the region R, which can be safely pruned as MkOLS search only considers the locations outside the region R. On the other hand, if $mindist(B, R) > d_c$, then for any location $b \in B$, its optimal set is empty, i.e., $S_b = \emptyset$. This is because $\forall a \in D_A \wedge a \in R$ and $\forall b \in B, dist(a, b) \geqslant mindist(B, R) > d_c$. Since M$k$OLS retrieval only considers the locations with positive optimality score, B can be safely pruned. The proof completes. □

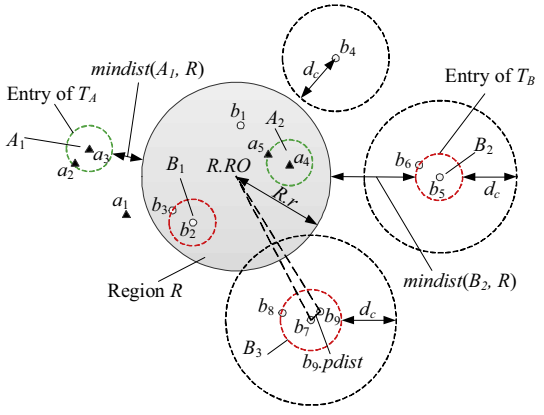


Fig. 3. Illustration of pruning rules.

Rule 4. Given an entry $B \in T_B$ and its parent entry B_p, if $dist(B_p.o, R.o) + B.pdist + B.r - R.r \leqslant 0$ or $|dist(B_p.o, R.o) - B.pdist| - B.r - R.r > d_c$, then B can be safely discarded.

Proof. It is correct according to Rule 3 and the triangle inequality utilizing the parent entry B_p, and thus omitted. □

Similarly, both Rules 3 and 4 can prune away the entries in T_B. Rule 3 is based on certain distance measures between B and R, while Rule 4 is based on distance measures between B's parent entry B_p and R. Take Fig. 3 as an example. Based on Rule 3, all the entries in T_B that are located inside R, i.e., b_1 and B_1, can be pruned because of $(maxdist(B, R.o) - R.r) \leqslant 0$; the entries B_2 and b_4 can be pruned away due to $mindist(B, R) > d_c$. In addition, take b_9 as an example. Using Rule 4, it can be discarded as $(|dist(B_3.o, R.o) - b_9.pdist| - R.r) > d_c$.

The reason we develop the above rules is to filter out the objects in T_A that cannot affect the optimality score of any answer location, and filter out the locations in T_B that cannot become any answer location. In order to distinguish those filtered entries from the remaining entries, we introduce the concept of *qualified entry*, as formally defined in Definition 4. As an example, in Fig. 3, A_2 is a *qualified object entry*, and B_3 is a *qualified location entry*.

Definition 4 (*Qualified entry*). An object (leaf or non-leaf) entry in T_A that cannot be discarded by Rules 1 and 2 is called a *qualified object entry*; and a location (leaf or non-leaf) entry in T_B that cannot be pruned by Rules 3 and 4 is referred to as a *qualified location entry*.

Algorithm 1. Baseline Algorithm (BL).

Input: k, R, d_c, T_A, T_B
Output: *Res*
1: initialize max-heaps $H(k)$ and H_A, stacks st_B, st, and *temp*, structure *LM*
2: $H_A = \{a_i | a_i \in T_A \wedge a_i \in R\}$ //Rule 1/2
3: push the root entries of T_B outside R into st //Rule 3/4
4: **while** $H_A \neq \emptyset$ **do**
5: entry a = H_A.deHeap()
6: $st_B = \emptyset$
7: Traverse-$D_B(a, R, d_c, st_B, st, temp)$
8: $st = temp \cup st_B$ and $temp = \emptyset$
9: **while** $st_B \neq \emptyset$ **do**
10: $(b, dist(a, b)) = st_B.pop()$
11: $LM[b]$.append$(a, dist(a, b))$
12: **for** each b in *LM* **do**
13: calculate $b.OPT$ by Definition 2
14: H.insert(b)
15: $Res = H$ and **return** *Res*

Function: Traverse-$D_B(a, R, d_c, st_B, st, temp)$
16: **while** $st \neq \emptyset$ **do**
17: entry e = st.pop()
18: **if** e is leaf entry **then**
19: **if** $dist(e, a) \leqslant d_c$ **then**
20: st_B.push(e)
21: **else**
22: **for** each child entry $e_i \in e$ **do**
23: push qualified location entry e_i into st, and **break** //Rule 3/4
24: push e_i which is located outside R into *temp* //Rule 3/4

3.3. Baseline algorithm

Our baseline algorithm (BL) fully utilizes the pruning rules presented above, which shares the same idea as RRB algorithm [16] (that performs the best in [16]). Nonetheless, the differences between BL and RRB are as follows: (1) BL employs the metric index M-tree while RRB uses R-tree and (2) BL utilizes Rules 2 and 4 to further improve query efficiency. In a word, BL is actually a simple adaption of RRB with some additional improvements.

The basic idea of BL is as follows. It first locates all the qualified objects via Rules 1 and 2, maintained by a max-heap H_A. Then, for each qualified object a in H_A, it finds all the qualified locations whose optimality scores can be affected by a via Rules 3 and 4, maintained by a stack st_B. Next, it adopts a brute-forth approach to evaluate the optimality score for each qualified location. Finally, the k locations with the highest optimality score are returned.

The pseudo-code of BL algorithm is depicted in Algorithm 1. First of all, BL initializes a max-heap H with the capacity k to store temporary results, a max-heap H_A to keep all the qualified objects, a stack st_B holding all the qualified locations whose optimality scores may be affected by a specified object a, two stacks st and $temp$ holding the qualified locations (i.e., those locations outside R), and a map structure LM to preserve the locations and their corresponding optimal sets (line 1). Note that, we need two stacks st and $temp$ to store the qualified location entries, with one serving as the working stack and the other serving as the auxiliary stack in order to enable reuse. Then, it preserves all the qualified objects that are not pruned away by Rules 1 and 2 in H_A (line 2), and pushes the root entries of T_B that are outside or intersected with R into st (line 3). Thereafter, it evaluates all the qualified objects maintained in H_A one by one. For each evaluated object $a \in H_A$, the evaluation has two steps. First, it invokes a function Traverse-D_B to evaluate the impact of a on all the qualified locations, i.e., those locations maintained by st (line 7). As shown in lines 16–24, Traverse-D_B inserts all the locations whose optimality scores could be affected by a into st_B. Meanwhile, it still keeps all the qualified location entries, i.e., those intersected with or located outside R, via $temp$ for the reuse later. Second, it updates LM based on st_B returned by the function Traverse-D_B. To be more specific, the structure LM maintains, for each potential location b, a list of objects that contribute to b's optimal set via $LM[b]$. When a is confirmed to affect b's optimality score, we insert a into $LM[b]$. In other words, once all the qualified objects maintained by H_A are evaluated, $LM[b]$ has the set of objects that form its optimal set, i.e., $LM[b] = S_b$. We then derive the optimality scores of all the locations b in LM, and the top-k locations with the highest optimality scores form the final result set (lines 12–15).

Example 1. We now illustrate how BL algorithm answers MkOLS search using our running example depicted in Fig. 2, with corresponding M-trees over D_A and D_B illustrated in Fig. 4 and $k = 1$. Initially, BL initializes H_A to the set of qualified objects (i.e., $H_A = \{a_7, a_6, a_1, a_5, a_4, a_2, a_3\}$) with the objects sorted based on descending order of their $mindist$ to the center of R. Note that, object a_8 is pruned away by Rule 1 as $mindist(a_8, R) > 0$. Stack st has its initial entries $\{B_1, B_2\}$. Then, it evaluates the entries of H_A one by one. First, a_7 is evaluated, with details shown in Table 2. Using the function Traverse-D_B, (i) entry B_4 is discarded by Rule 2 because of $mindist(B_4, a_7) > d_c$, but preserved in $temp$ for the reuse later; (ii) location b_1 whose optimality score cannot be affected by a_7 is inserted into $temp$; (iii) other locations are added to $st_B = \{b_2, b_5, b_6, b_7, b_8\}$, which takes a_7 in their optimal sets. After the evaluation, we have $st_B = \{b_2, b_5, b_6, b_7, b_8\}$, $temp = \{B_4, b_1\}$, $LM[b_2] = \{(a_7, dist(b_2, a_7))\}$, $LM[b_5] = \{(a_7, dist(b_5, a_7))\}$, $LM[b_6] = \{(a_7, dist(b_6, a_7))\}$, $LM[b_7] = \{(a_7, dist(b_7, a_7))\}$, and $LM[b_8] = \{(a_7, dist\ (b_8, a_7))\}$.

Second, BL pops a_6 from H_A, reuses st, and finds all the qualified locations $\{b_7, b_8\}$ that can be affected by a_6 with $LM[b_2] = \{(a_7, dist(b_2, a_7))\}$, $LM[b_5] = \{(a_7, dist(b_5, a_7))\}$, $LM[b_6] = \{(a_7, dist(b_6, a_7))\}$, $LM[b_7] = \{(a_7, dist(b_7, a_7)), (a_6, dist\ (b_7, a_6))\}$, and $LM[b_8] = \{(a_7, dist\ (b_8, a_7)), (a_6, dist(b_8, a_6))\}$. The algorithm proceeds in the same manner until the heap H_A becomes empty. Table 3 depicts the final contents of LM. Next, BL computes, for every location in LM, its optimality score according to Definition 2. In the end, b_8 that has the highest optimality score is returned as the optimal location.

4. MkOLS query processing

Although BL algorithm fully utilizes our presented pruning rules, the performance is highly dependent on the number of qualified objects/locations. No matter how small k is, it has to evaluate all the qualified objects/locations. For instance, in Example 1, even though only *one* optimal location is required, BL has to evaluate the optimality scores for *five* qualified

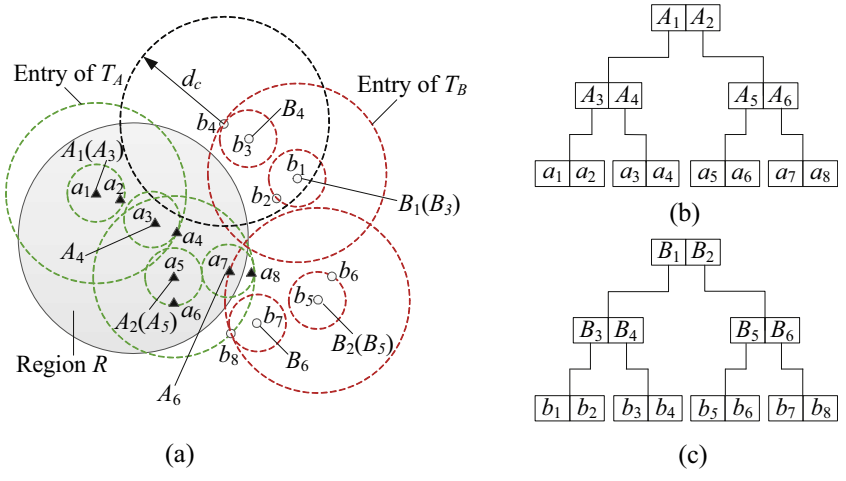


Fig. 4. Illustration of the M-trees on D_A and D_B.

Table 2
Illustration of Traverse-D_B for a_7.

Operation	st	st_B	$temp$
Initialization	B_1, B_2	$\emptyset$	$\emptyset$
Visit B_1	B_3, B_2	$\emptyset$	B_4
Visit B_3	b_2, B_2	$\emptyset$	B_4, b_1
Visit b_2	B_2	b_2	B_4, b_1
Visit B_2	B_5, B_6	b_2	B_4, b_1
Visit B_5	b_5, b_6, B_6	b_2	B_4, b_1
Visit b_5	b_6, B_6	b_5, b_2	B_4, b_1
Visit b_6	B_6	b_6, b_5, b_2	B_4, b_1
Visit B_6	b_7, b_8	b_6, b_5, b_2	B_4, b_1
Visit b_7	b_8	b_7, b_6, b_5, b_2	B_4, b_1
Visit b_8	$\emptyset$	b_8, b_7, b_6, b_5, b_2	B_4, b_1

locations (b_2, b_5, b_6, b_7, and b_8), incurring *high* I/O and CPU costs. In order to develop a k-sensitive algorithm with better performance, we propose an early termination technique based on the estimation of the optimality score and then present an efficient search algorithm to support MkOLS retrieval.

4.1. Estimation-based early termination

According to Definition 2, for a location b with its optimal set S_b, $b.OPT$ is within the range of $(|S_b| - 1, |S_b|]$. Consequently, given two locations b_i and b_j, if $|S_{b_i}| > |S_{b_j}|$, it is certainly $b_i.OPT > b_j.OPT$. As the calculation of the exact optimality score (i.e., $b_i.OPT$) is more expensive than finding the size of its optimal set (i.e., $|S_{b_j}|$), we would like to take $|S_{b_j}|$ as an estimation (i.e., the *upper bound*) of $b_i.OPT$. In the following, we first explain how to estimate the optimality score for a specified location, and then present an early termination condition based on the estimated optimality score.

Since the optimality score considers the cardinality of the optimal set but not the real content of the optimal set, we only care about how many objects are there in an optimal set. Therefore, we present the concept of *object count* in Definition 5, based on which the *estimated optimality score* is developed, as presented in Definition 6.

Definition 5 (*Object count*). Given an intermediate entry e of a tree T built on an object set O, let $e.S_o$ be the set of objects that are contained in the subtree (sub_e) rooted by e, i.e., $e.S_o = \{o | o \in O \wedge o \in sub_e\}$, its cardinality is referred to as the *object count* of e, denoted as $e.|S_o|$.

Definition 6 (*Estimated optimality score*). Assume that the current view of T_A in the memory st is $V(T_A) = \{A_1, A_2, \ldots, A_m, a_1, a_2, \ldots, a_n\}$, where A_i is an intermediate entry and a_i is an object. For a location entry e_b representing either a set of locations or a real location, its *estimated optimality score* is defined as:

$$e_b.EST = \sum_{A_i \in V(T_A) \wedge mindist(A_i, e_b) \leqslant d_c} A_i.|S_o| + |\{a_j | a_j \in V(T_A) \wedge mindist(a_j, e_b) \leqslant d_c\}|. \quad (1)$$

Given a tree T_A rooted at R, it is explored during the process of MkOLS search. Consequently, we use $V(T_A)$ to represent the current view of T_A in the form of a set of intermediate entries and objects. Consider our running example. Before the process of MkOLS retrieval, $V(T_A) = \{A_1, A_2\}$. Assume that entry A_1 is expanded, then $V(T_A) = \{A_3, A_4, A_2\}$ and so on. Given a current view of T_A, the estimated optimality score of a location entry can be calculated based on Eq. (1). Notice that, Eq. (1) treats intermediate entries and objects in $V(T_A)$ differently. For intermediate entries A_i, it utilizes *object count* to count the number of objects contained in A_i that could contribute to the optimal set of the location b; for objects a, it evaluates the objects

Table 3
Illustration of *LM* in BL.

Key	Value
b_1	$\emptyset$
b_2	$(a_7, dist(b_2, a_7))$
b_3	$\emptyset$
b_4	$\emptyset$
b_5	$(a_7, dist(b_5, a_7))$
b_6	$(a_7, dist(b_6, a_7))$
b_7	$(a_7, dist(b_7, a_7)), (a_6, dist(b_7, a_6)), (a_5, dist(b_7, a_5))$
b_8	$(a_7, dist(b_7, a_7)), (a_6, dist(b_7, a_6)), (a_5, dist(b_7, a_5))$

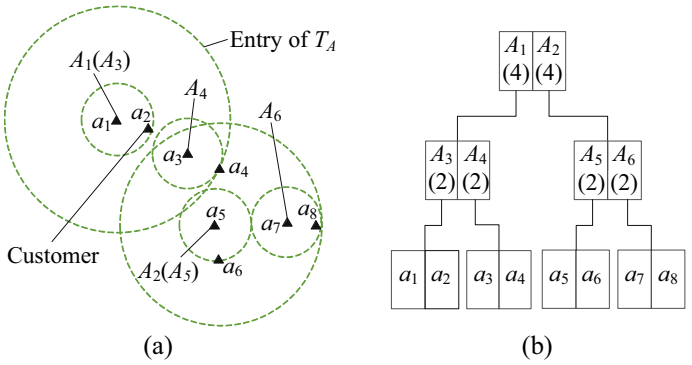


Fig. 5. Illustration of the COM-tree on D_A.

directly (i.e., $mindist(a,b) \leqslant d_c$). In other words, when the view is in high level, the estimated optimality score tends to be very loose as it assumes all the objects in an entry A_i with $mindist(A_i,b) \leqslant d_c$ contributing to b's optimal set while actually only some of them contributing to b's optimal set. Since we expand our view of the tree, the estimated optimality score gets closer to the real optimality score.

The reason we introduce the concept of estimated optimality score is to derive the upper bound of the optimality score for a real location or a set of locations, as stated in Lemma 1. Based on the estimation, we can easily compare the optimality score between two locations, or even compare the optimality score of a set of locations (in the form of an intermediate entry) with the optimality score of another location, as stated in Lemma 2 and 3, respectively.

Lemma 1. *Given a qualified location intermediate entry* $B \in D_B, \forall b \in B, b.OPT \leqslant B.EST$.

Proof. Without loss of generality, we assume $V(T_A) = \{A_1, A_2, \ldots, A_j\}$ where $A_i \in V(T_A)$ refers to an intermediate entry or an object in T_A, and $V(T_A)_{est} = \{A_i | A_i \in V(T_A) \wedge mindist(A_i,b) \leqslant d_c\}$. Given a location $b \in B, \forall a \in S_b, \exists A_i \in V(T_A)$ such that $a \in A_i$. Based on the definition of optimal set, $dist(a,b) \leqslant d_c$. As $a \in A_i, mindist(A_i,b) \leqslant dist(a,b) \leqslant d_c$ and hence $A_i \in V(T_A)_{est}$. In other words, we have $S_b \subseteq V(T_A)_{est}$. Since $B.EST = \sum_{A_i \in V(T_A)_{est}} A_i.|S_o|$ and $b.OPT \leqslant |S_b|, b.OPT \leqslant B.EST$ holds. The proof completes. □

Lemma 2. *Given two qualified locations* $b_i, b_j \in D_B$, *if* $b_i.EST \leqslant b_j.OPT$, *then* $b_i.OPT \leqslant b_j.OPT$.

Proof. Based on Lemma 1, for a specified qualified location $b_i \in D_B, b_i.OPT \leqslant b_i.EST$. If $b_i.EST \leqslant b_j.OPT$, then $b_i.OPT \leqslant b_i.EST \leqslant b_j.OPT$. The proof completes. □

Lemma 3. *Given a qualified location entry B and a qualified location object b, if* $B.EST \leqslant b.OPT$, *then for* $\forall b_i \in B, b_i.OPT \leqslant b.OPT$.

Proof. Based on Lemma 1, $\forall b_i \in B, b_i.OPT \leqslant B.EST$. If we know that $B.EST \leqslant b.OPT$, then $\forall b_i \in B, b_i.OPT \leqslant B.EST \leqslant b.OPT$. The proof completes. □

Based on the aforementioned lemmas, we propose to explore the entries in T_B based on their estimated optimality scores, as stated in Theorem 1. According to this new exploring order, it is not necessary to explore all the qualified locations and hence improve MkOLS search accordingly.

Theorem 1. *Assume that we explore entries e of* T_B *in descending order of their estimated optimality scores (i.e., e.EST), and maintain top-k locations with the highest optimality score in a candidate set C. Let e be the current evaluated entry. If* $|C| = k$ *and* $e.EST \leqslant MIN_{b \in C} b.OPT$, *entry e and all the remaining unexplored entries cannot contain any answer location for MkOLS retrieval.*

Proof. Assume that the above statement is not valid. In other words, there is at least one location $b \in e_i$ with $e_i.EST \leqslant e.EST$ is actually an answer location for a MkOLS query. As b is an answer location, $b.OPT > MIN_{b \in C} b.OPT$. On the other hand, according to Lemma 1, $b.OPT \leqslant e_i.EST$. Since $e_i.EST \leqslant e.EST, b.OPT \leqslant e_i.EST \leqslant e.EST$. In other words, $MIN_{b \in C} b.OPT < b.OPT \leqslant e_i.EST \leqslant e.EST$ which contradicts with the fact that $e.EST \leqslant MIN_{b \in C} b.OPT$. Consequently, our assumption is invalid, and the proof completes. □

In order to enable this new exploring order, we propose a variant of M-tree, termed as *COUNT M-tree* (COM-tree), which can facilitate the estimation of optimality score according to Definition 6. Specifically, for each intermediate entry e, we add the object count $e.|S_o|$ to the original M-tree. For ease of understanding, the COM-tree on D_A is illustrated in Fig. 5(b), where

the number in every intermediate entry e refers to $e.|S_o|$. Note that, these numbers are obtained during the construction of COM-tree. In addition, since the main structure for M-tree is not changed, our presented distance-based pruning rules (presented in Section 3.2) are still applicable to COM-tree.

4.2. Estimation-based algorithm

Based on Theorem 1 and COM-tree, we develop our second algorithm, namely, *estimation-based algorithm* (EB), to answer the M*k*OLS query. For EB, we build a COM-tree T_A on D_A and an M-tree T_B on D_B, respectively. The location entries in T_B are evaluated based on their estimated optimality scores, i.e., the entries e_b with larger $e_b.EST$ are processed earlier in order to apply Theorem 1.

To estimate the optimality scores for those location entries, T_A is traversed based on the breadth-first paradigm. In particular, when a location entry e_b is processed, we traverse the entries in T_A that may affect $e_b.EST$ down one level, in order to refine the estimation precision. Theorem 1 serves as an early termination condition. If it is satisfied, the algorithm terminates immediately and returns the result; otherwise, it proceeds to evaluate entries in T_B.

Algorithm 2. Estimation-based Algorithm (EB).

Input: k, R, d_c, T_A, T_B
Output: *Res*
1: initialize stacks st, st_A, and *temp*, max-heap H_B, and min-heap $H(k)$
2: push root entries in T_A located inside R into st
3: calculate *EST* for qualified root entries in T_B, and push them into H_B
4: **while** $e_b = H_B$.pop() $\neq \emptyset$ **do**
5: **if** $|H| = k \wedge e_b.EST \leqslant H$.top().*OPT* **then**
6: **return** H //Theorem 1
7: **while** $st \neq \emptyset$ **do**
8: pop entry e_a from st
9: **if** $mindist(e_a, e_b) \leqslant d_c \wedge e_a$ is a non-leaf entry **then**
10: push its qualified children into st_A //Rule 1/2
11: **else if** $mindist(e_a, e_b) \leqslant d_c \wedge e_a$ is an object **then**
12: push e_a into st_A
13: **else**
14: push e_a into *temp*
15: **if** e_b is a non-leaf location **then**
16: **for** each qualified child e_{b_j} of e_b **do**
17: calculate $e_{b_j}.EST$ using st_A //Rule 3/4
18: H_B.insert($e_{b_j}, e_{b_j}.EST$)
19: **else**
20: **for** each entry e_a in st_A **do**
21: **if** e_a is a non-leaf entry and $maxdist(e_a, e_b) > d_c$ **then**
22: replace e_a with all qualified objects contained in e_a that affect the optimality score of e_b
23: compute $e_b.OPT$ using st_A
24: H.insert($e_b, e_b.OPT$)
25: $st = st_A \cup temp$ and $st_A = temp = \emptyset$
26: $Res = H$ and **return** *Res*

Algorithm 2 depicts the pseudo-code of EB. To begin with, EB initializes all the important data structures (lines 1–3). H_B is a max-heap to store fetched qualified entries in T_B, sorted in descending order of their estimated optimality scores. Note that, if two entries have the same estimated optimality score, the entry e with smaller $mindist(e, R)$ is evaluated earlier. Initially, it contains the qualified root entries of T_B. H is a min-heap with size of k to maintain those retrieved locations with the highest optimality score. The top entry of H tells the minimum optimality score of current candidates (i.e., $MIN_{b \in C} b.OPT$). Note that, when H contains less than k locations, $MIN_{b \in H} b.OPT$ returns *zero*. Two stacks *st* and *temp*, one as a working stack and the other as an auxiliary stack, maintain all the qualified object entries. Initially, *st* contains the root entries of T_A that are located inside R, and *temp* is empty. Similar as stacks *st* and *temp* used in BL algorithm, they are to implement reuse technique so that we only need to access object entries in T_A once. st_A is also a stack to preserve all the qualified object entries that might affect the optimality score of a certain location b.

After initialization, it evaluates location entries based on descending order of their estimated optimality scores (lines 4–25). To be more specific, it pops out the top entry e_b for evaluation. If the condition listed in Theorem 1 is satisfied, the algorithm can be terminated earlier by returning the current set of candidate locations maintained by H (lines 5–6). Otherwise,

Table 4
Illustration of EB.

Operation	H_B	st	H
Initial	$B_1(8), B_2(8)$	A_1, A_2	$\emptyset$
Visit B_1	$B_2(8), B_4(4), B_3(2)$	A_3, A_4, A_5, A_6	$\emptyset$
Visit B_2	$B_6(4), B_4(4), B_3(2), B_5(1)$	A_3, A_4, A_5, a_7, a_8	$\emptyset$
Visit B_6	$b_7(4), b_8(4), B_4(4), B_3(2), B_5(1)$	$A_3, a_3, a_4, a_5, a_6, a_7, a_8$	$\emptyset$
Visit b_7	$b_8(4), B_4(4), B_3(2), B_5(1)$	$A_3, a_3, a_4, a_5, a_6, a_7, a_8$	b_7
Visit b_8	$B_4(4), B_3(2), B_5(1)$	$A_3, a_3, a_4, a_5, a_6, a_7, a_8$	b_8
Visit B_4	$B_3(2), B_5(1)$	$A_3, a_3, a_4, a_5, a_6, a_7, a_8$	b_8

we expand the current view of T_A preserved in st down by one level to calculate the estimated optimality score of e_b's child entries (lines 7–14). In particular, (i) for the non-leaf entry e_a with $mindist(e_a, e_b) \leqslant d_c$, we push all its qualified children into st_A, according to Rule 1/2; (ii) for the leaf entry e_a with $mindist(e_a, e_b) \leqslant d_c$, we push it into st_A; and (iii) for the (leaf or non-leaf) entry with $mindist(e_a, e_b) > d_c$, we maintain it in $temp$. Thereafter, if e_b is a non-leaf location, we can derive the estimated optimality scores of all the qualified child entries of e_b and insert them back to H_B for further evaluation (lines 15–18). Note that, in BL algorithm, we access T_A down to the leaf level, and evaluate directly the impact of each qualified object $a \in T_A$ on location b's optimality score. However, in EB algorithm, the exploration of T_A is triggered by the evaluation of qualified location entries maintained by H_B. Initially, we only know the root entries of T_A. As entries in H_B are evaluated, entries of T_A are explored, and the view of T_A is expanded. This is because $e_b.EST$ can be evaluated based on a view of $V(T_A)$ (maintained by s_t), from the most abstract view with $V(T_A)$ only containing root entries to the most detailed view with each entry in $V(T_A)$ representing an object. This hierarchical exploration of T_A and T_B can effectively prevent the exploration of those entries that only contain locations with either zero optimality score or very small optimality score, and hence contributes to the improvement of search performance. In addition, we also want to highlight that st_A might contain non-leaf entries e_a when evaluating OPT for a location e_b (lines 21–22). If $maxdist(e_a, e_b) > d_c$, we traverse down to get all qualified objects in the subtree rooted by e_a to compute e_b's optimality score. After computing OPT for a location e_b using st_A, we then update the result heap H accordingly (lines 23–24). Finally, EB returns the final result.

Example 2. Back to our running example depicted in Fig. 4 with $k = 1$. Initially, $st = \{A_1, A_2\}$, and we compute the estimated optimality score for root entries B_1 and B_2 of T_B, with $B_1.EST = B_2.EST = A_1.|S_o| + A_2.|S_o| = 8$. As B_1 is closer to $R.o$ than B_2, H_B is set to $\{B_1(8), B_2(8)\}$. After the initialization, we continuously evaluate the location entries popped out from H_B. First, B_1 is evaluated. As both A_1 and A_2 in st may affect $B_1.EST$, we expand them, with their child entries maintained by st_A ($= \{A_3, A_4, A_5, A_6\}$). We then derive the estimated optimality score of B_1's child entries based on the content of st_A, with $B_3.EST = A_6.|S_o| = 2$ and $B_4.EST = A_4.|S_o| + A_6.|S_o| = 4$. Then, H_B is updated to $\{B_2(8), B_4(4), B_3(2)\}$. Similarly, we continue this process to evaluate the entries popped from H_B. When b_7 is visited, we first push all qualified entries maintained in st that will affect the optimality score of b_7 into $st_A = \{a_4, a_5, a_6, a_7\}$. As b_7 is a leaf entry, we compute $b_7.OPT$ using st_A, and insert b_7 into H. Next, we evaluate location b_8 similarly, and update $H = \{b_8\}$ as $b_8.OPT < b_7.OPT$. In the following, EB proceeds to evaluate entries in H_B until the early termination satisfies as $B_3.EST < b_8.OPT$. Finally, the optimal location b_8 is returned. Table 4 depicts the detailed steps of EB. □

4.3. Discussion

In the sequel, we first prove the correctness of the proposed algorithms, and then analyze their performance.

Lemma 4. *Both the proposed algorithms (i.e., BL and EB) return exactly the actual MkOLS query result, i.e., both algorithms have no false negatives, no false positive, and the returned result set contains no duplicate objects.*

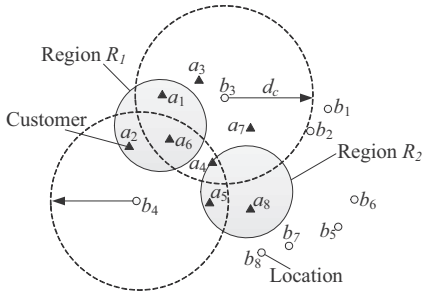


Fig. 6. Illustration of MkOLS$_{MR}$ and MkOLS$_{NR}$ queries.

Proof. First, no result is missed (i.e., *no false negatives*) as only unqualified (leaf and non-leaf) entries are pruned by our developed Rules and Theorem 1. Second, all locations that can become the optimal locations are evaluated and verified against other qualified locations in T_B to ensure *no false positive*. Third, *no duplicate objects* are generated because each location is evaluated at most once and are popped right after evaluation. The proof completes. □

It is worth noting that, EB outperforms BL significantly. This is because BL needs to evaluate all the qualified locations; while EB evaluates the locations in descending order of their estimated optimality scores, and it can terminates the evaluation once it retrieves at least k locations and is certainly that all the unexamined locations have their optimality scores smaller than that of retrieved locations. In other words, it is very likely that EB only evaluates some, but not all, qualified locations. As to be demonstrated in Section 6, EB avoids lots of unnecessary location evaluations which significantly cuts down the query cost.

5. Extension

In this section, we show the flexibility and extensibility of our proposed EB algorithm by studying two interesting variations of MkOLS queries, namely, MkOLS search with *multiple* constrained regions (MkOLS$_{\mathrm{MR}}$), and MkOLS search with non-constrained region (MkOLS$_{\mathrm{NR}}$).

5.1. MkOLS$_{\mathrm{MR}}$ search

According to the definition of MkOLS retrieval, only one constrained region R is considered. However, in some real applications, there may exist several constrained regions. Take Application 1 as an example. If the new Pizza Hut branch may serve a few residential areas while it cannot be located inside any of them, we have to take into account multiple constrained regions. In view of this, a useful variant of MkOLS queries, i.e., MkOLS search with multiple constrained regions (i.e., MkOLS$_{\mathrm{MR}}$), is proposed.

Definition 7 (*MkOLS$_{\mathrm{MR}}$ search*). Given an object set D_A, a location set D_B, a critical distance d_c, n constrained regions $R_i (1 \leqslant i \leqslant n)$, and an integer k ($\geqslant 1$) in a metric space, an MkOLS$_{\mathrm{MR}}$ query finds the k locations in D_B having the maximal optimality scores among all the locations located outside R_i.

Note that, the optimality score of a location for MkOLS$_{\mathrm{MR}}$ retrieval is similar as its original setting presented in Definition 2. Nevertheless, MkOLS$_{\mathrm{MR}}$ search takes n regions into consideration. Therefore, given an object $a \in D_A$ and a location $b \in D_B$, if it is within one of those n regions and meanwhile has its distance to b bounded by d_c, the object a contributes to the location b's optimal set S_b^{MR}, i.e., $S_b^{MR} = \{a_i | a_i \in D_A \wedge \exists j \in [1,n], a_i \in R[j] \ \wedge \ dist(a_i, b) \leqslant d_c\}$. Here, $dist(a_i, b)$ is a metric distance and a_i, b can be any type of objects in the metric space.

An example of MkOLS$_{\mathrm{MR}}$ query with two constrained regions R_1 and R_2 is shown Fig. 6. Suppose $k = 1$, b_4 is the optimal location since it covers the most qualified objects in R_1 and R_2, with $S_{b_4}^{MR} = \{a_2, a_4, a_5, a_6\}$. The EB algorithm can be easily extended to the *estimation-based algorithm for* MkOLS$_{\mathrm{MR}}$ (EBM) for answering MkOLS$_{\mathrm{MR}}$ search. Notice that, there are two major changes. First, pruning rules for objects (i.e., Rules 1 and 2) need to consider the distance between the objects and all n regions. An object entry A can be pruned only when its distances to all n regions satisfy the condition. Second, pruning rules for locations (i.e., Rules 3 and 4) also need to consider the distance between a location entry and all n regions.

5.2. MkOLS$_{\mathrm{NR}}$ search

Both MkOLS and MkOLS$_{\mathrm{MR}}$ queries assume there is at least one constrained region. However, MkOLS retrieval without any constrained region, namely, MkOLS$_{\mathrm{NR}}$ search, also have a application base. Again, consider the Pizza Hurt restaurants example. If a new Pizza Hut restaurant can be located in any location, it can be supported by MkOLS$_{\mathrm{NR}}$ retrieval.

Definition 8 (*MkOLS$_{\mathrm{NR}}$ search*). Given an object set D_A, a location set D_B, a critical distance d_c, and an integer k ($\geqslant 1$) in a metric space, an MkOLS$_{\mathrm{NR}}$ query finds the k locations in D_B having the maximal optimality scores among all the locations.

Note that, the optimality score considered by MkOLS$_{\mathrm{NR}}$ search is also similar as the original optimality score considered by MkOLS retrieval. The only difference is that, since there is *no* constrained region R, the optimal set $S_{b_j}^{NR}$ for b_j ($\in D_B$) includes all the objects whose distances to b_j bounded by d_c, i.e., $S_{b_j}^{NR} = \{a_i | a_i \in D_A \wedge dist(a_i, b_j) \leqslant d_c\}$. Here, $dist(a_i, b_j)$ is a metric distance, and a_i, b_j can be any type of objects in metric spaces.

Table 5
Statistics of the data sets used.

Dataset	Cardinality	Dimensionality	Metric
LA	500 K	2	L_1-norm
Uniform	[250 K, 4 M]	[2,5]	L_2-norm
Zipf	[250 K, 4 M]	[2,5]	L_∞-norm

Table 6
Parameter ranges and default values.

Parameter	Settings	Default
k	1, 4, 16, 64, 256	16
R (% of full space)	1, 2, 4, 8, 16	4
$d_c(*1000)$	0.6, 0.8, 1, 1.2, 1.4, 1.6	1
Dimensionality	2, 3, 4, 5	2
Cardinality (M)	0.25, 0.5, 1, 2, 4	1
Number of constrained regions	0, 1, 2, 3, 4, 5	1

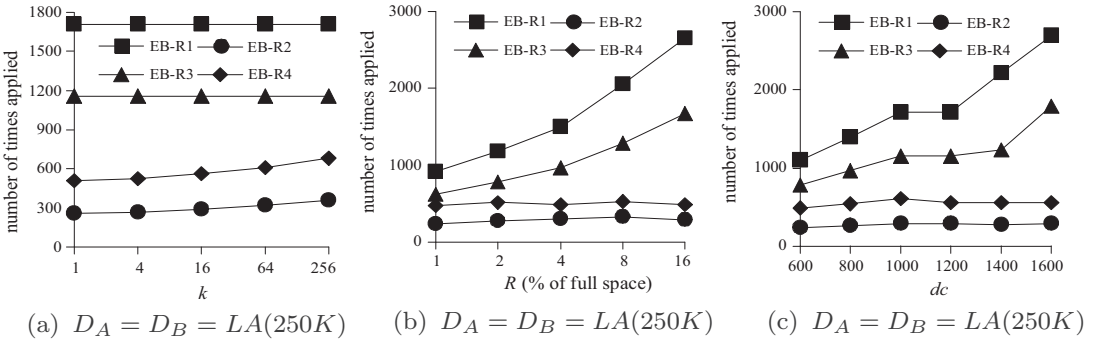


Fig. 7. Pruning rule efficiency vs. k, R, and d_c.

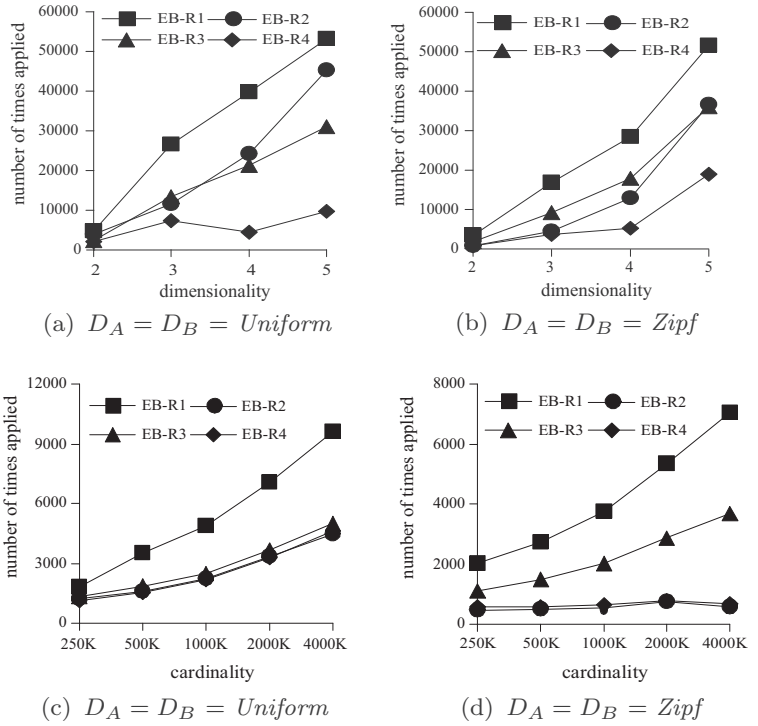


Fig. 8. Pruning rule efficiency vs. dimensionality and cardinality.

Consider the example depicted in Fig. 6 again. If we remove the constrained regions R_1 and R_2, $S^{NR}_{b_3} = \{a_1, a_3, a_4, a_6, a_7\}$ and $S^{NR}_{b_4} = \{a_2, a_4, a_5, a_6\}$. If $k = 1$, MkOLS$_{\text{NR}}$ search will return b_3 as the answer location because it has the largest optimal set. Our EB algorithm can also be easily adjusted to the *estimation-based algorithm for* MkOLS$_{\text{NR}}$ (EBN) to tackle MkOLS$_{\text{NR}}$ retrieval. Since there is no constrained region, all the entries should be considered as qualified entries, i.e., all the four pruning rules developed in Section 3 cannot be applied here, as they are all based on the constrained region R.

6. Experimental evaluation

In this section, we experimentally evaluate the effectiveness of our developed pruning rules and the performance of our proposed algorithms for M*k*OLS search and its variants, using both real and synthetic datasets. All algorithms were implemented in C++, and all experiments were conducted on an Intel Core 2 Duo 2.93 GHz PC with 3 GB RAM.

We employ a real dataset *LA* containing 500 K downtown locations in Los Angeles. Specifically, we partition *LA* into two different datasets D_A and D_B to simulate three cases (i) $|D_A| < |D_B|$, (ii) $|D_A| = |D_B|$, and (iii) $|D_A| > |D_B|$. The distance between two points in *LA* is measured as L_1-norm, which is typically used as an approximation of the road network distance [29].

We also generate several synthetic datasets with dimensionality varying in the range of [2,5] and cardinality varying in the range of [250 K, 4 M], following Uniform and Zipf distributions. Table 5 summarizes the datasets used in our experiments. The coordinate of each point in *Uniform* datasets is generated uniformly along every dimension, and that of each point in *Zipf* datasets is generated according to a Zipf distribution with skew coefficient $\alpha = 0.8$. Without loss of generality, L_2-norm (i.e., the Euclidean distance) is used as the distance metric for *Uniform* datasets, and L_∞-norm is utilized to compute the distance between two points in *Zipf* datasets. Note that, for all datasets, every dimension in the data space is normalized to [0, 10,000]. All datasets are indexed by either COM-trees (presented in Section 4.1) or M-trees [9], with a page size of 4096 bytes.

We study the performance of the proposed algorithms under various parameters, which are listed in Table 6. Note that, in each experiment, only one factor varies, whereas the others are fixed to their default values. In addition, the center of region R is generated randomly, and the region R is bounded by the data space.

The main performance metrics include the query cost, the number of page accesses (*PA*), the number of locations calculated (*LC*), and *compdists*, which represents the number of distance computations. Here, the query cost refers to the sum of the I/O time and CPU time, where the I/O time is computed by charging 10 ms for each page access, as with [7]). Each reported value in the following diagrams is the average performance of 100 queries.

6.1. Effectiveness of pruning rules

The first set of experiments verifies the effectiveness of our presented pruning rules, in terms of the number of times rules are applied. We implement EB algorithm and count a rule applied if an object or an intermediate entry is pruned away by that rule. First, we study the impact of k, R, and d_c on the performance of pruning rules under the real dataset *LA* with $|D_A| = |D_B| = 250K$, as shown in Fig. 7. It is obvious that all the developed rules help to prune certain number of objects/

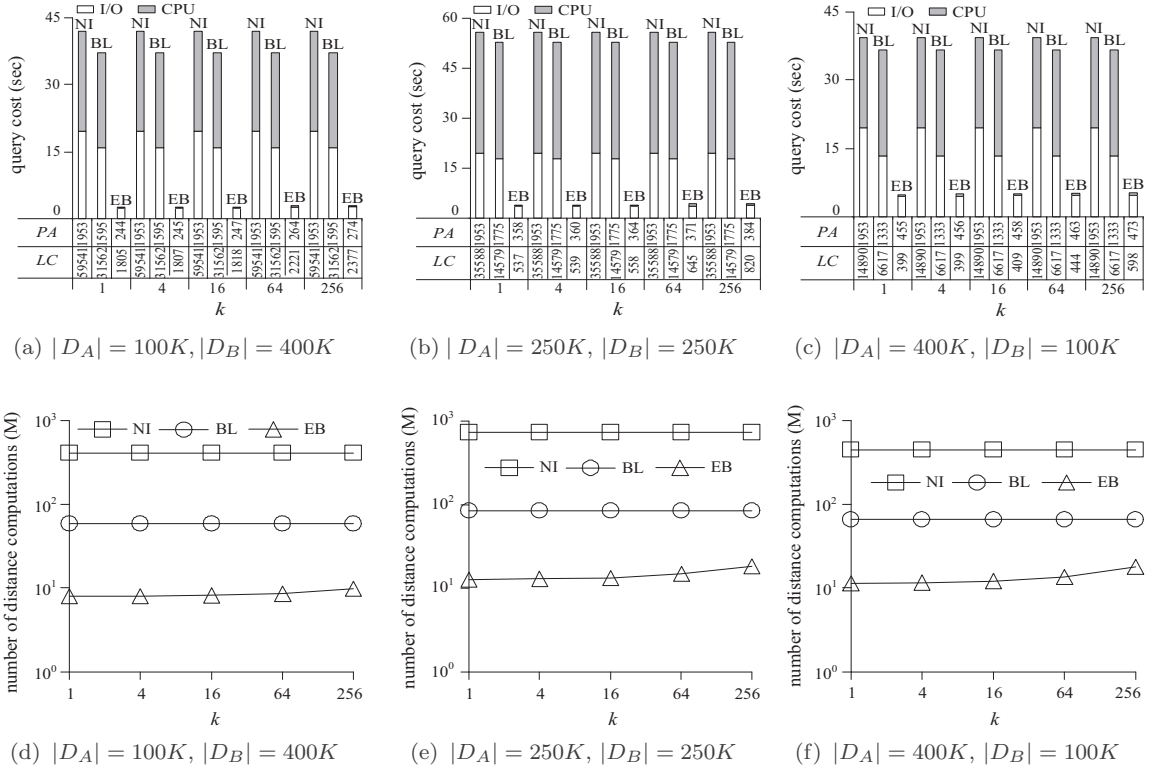
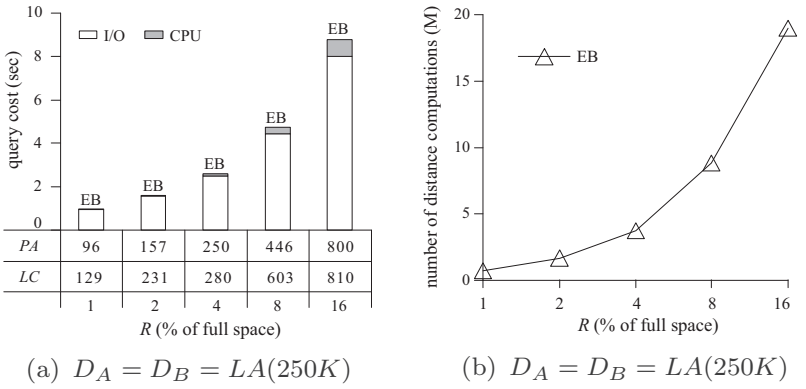
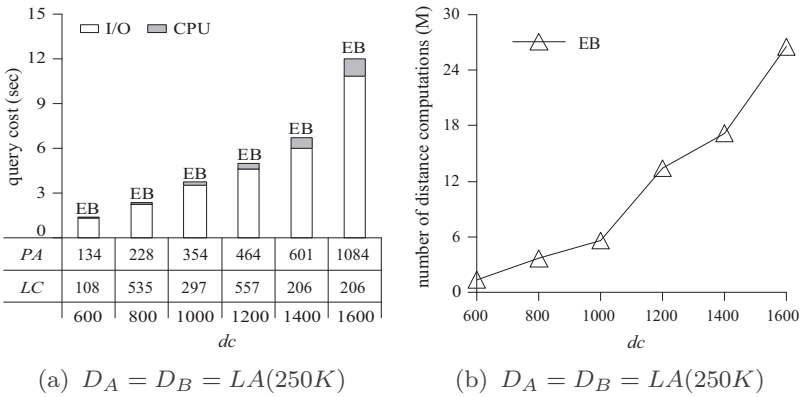


Fig. 9. M*k*OLS performance vs. *k*.



(a) $D_A = D_B = LA(250K)$ (b) $D_A = D_B = LA(250K)$

Fig. 10. M*k*OLS performance vs. *R*.



(a) $D_A = D_B = LA(250K)$ (b) $D_A = D_B = LA(250K)$

Fig. 11. M*k*OLS performance vs. d_c.

entries. Among three parameters, the parameters *R* and d_c play more significant role because all the rules are based on *R* and d_c. On the other hand, *k* does not affect the pruning power much. Note that, we skip the experimental results under real datasets with varying $|D_A|/|D_B|$, due to the similar experimental results and the space limitation.

In addition, we also explore the impact of dimensionality and cardinality respectively, using synthetic datasets, as depicted in Fig. 8. In general, the number of pruning applications increases as dimensionality grows. This is because, it is more difficult to prune high level entries as dimensionality ascends, resulting in a large number of low level entries pruning applications. As expected, the number of applications steadily increase with the growth of cardinality. The reason is that, when $|D_A|$ and $|D_B|$ increase, there are more records that may qualify the pruning rules.

6.2. Results on MkOLS search

The second set of experiments evaluates the performance of BL and EB algorithms in answering M*k*OLS queries. Since both BL and EB take advantage of the COM-tree structure to help to prune objects in batches. To give a comprehensive experimental study, we also consider the case where no indexes are available. In this case, a straightforward algorithm, so-called *NoIndex* (NI), can act as an ancillary solution. For simplicity, we assume that the data can be fully loaded into memory. In this case, NI begins with loading the objects and locations and pruning unnecessary ones (similar to Rules 1 and 3, but on single record level only). Then NI calculates the optimal score of each qualified location, and finally only the locations with top-*k* optimality scores are selected as the result. We study the influence of various parameters, including (i) the value of *k*, (ii) the size of constrained region *R*, (iii) the value of critical distance d_c, (iv) the dimensionality of datasets, and (v) the cardinality of datasets.

First, we investigate the effect of *k* on the efficiency of the algorithms based on real datasets, and report the results in Fig. 9. Notice that, the query cost is broken into I/O cost and CPU cost. The name on top of each bar refers to the specific algorithm; and *PA* and *LC* are shown at the bottom. Note that, as pointed out in Section 3.3, BL is actually a simple adaption of RRB algorithm [16] with some additional improvements. The real dataset *LA* is employed and three different dataset size combinations are considered, representing different cases for M*k*OLS queries: (i) $|D_A| < |D_B| : |D_A| = 100K, |D_B| = 400K$; (ii) $|D_A| \approx |D_B| : |D_A| = |D_B| = 250K$; and (iii) $|D_A| > |D_B| : |D_A| = 400K, |D_B| = 100K$.

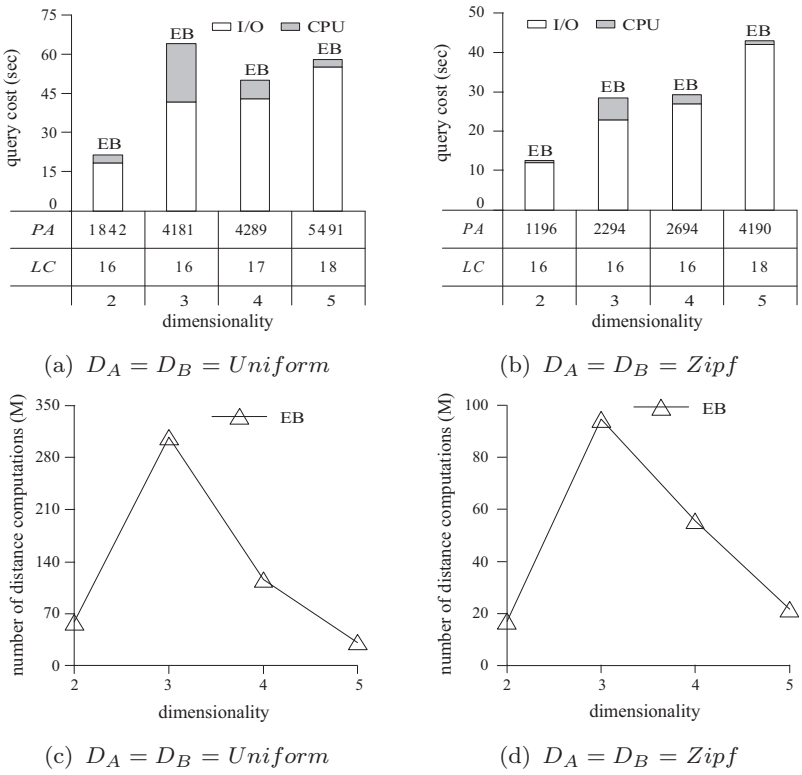


Fig. 12. M*k*OLS performance vs. dimensionality.

The first observation is that EB is several orders of magnitude better than NI and BL in all cases. The reason is that, NI has to fetch and evaluate all the qualified locations one by one in a brute-force way. As a result, the I/O and CPU costs of NI are significant. BL has been designed to reduce I/O overhead, but in the expense of higher CPU cost. EB, on the other hand, utilizes the estimation based early termination technique, which is efficient and sensitive to k. The second observation is that the performance of EB increases gradually with k, whereas the performance of both NI and BL is insensitive to k. This is because both NI and BL need to calculate the optimality score of each qualified location, regardless of the value of k. However, EB enables an early termination, and the value of k directly affects when the algorithm can be terminated. Since EB significantly outperforms both NI and BL (by factors) for all the cases including $|D_A| < |D_B|$, $|D_A| \approx |D_B|$, and $|D_A| > |D_B|$, we skip NI and BL but only report the performance of EB in the rest of experiments.

Then, we explore the influence of R on the efficiency of the algorithms using real data sets, with the results depicted in Fig. 10. As expected, the query cost and *compdists* of EB increase as R grows. The reason behind is that, there are more qualified objects and more qualified locations as R increases. Therefore, the algorithms require more page accesses and more query cost with the growth of R.

Fig. 11 plots the performance of EB as a function of d_c. As observed, the query cost and the number of distance computations (i.e., *compdists*) of EB increase with d_c, because the M*k*OLS search space grows as d_c ascends.

Next, we evaluate the impact of the dimensionality on the efficiency of EB algorithm based on synthetic datasets, and report the results in Fig. 12. The I/O cost ascends with dimensionality due to the reduced page capacity. A crucial observation is that the *compdists* decreases when dimensionality exceeds three, so does the CPU cost. This is because the object density drops as dimensionality grows [9], and thus, the number of objects that can contribute to the optimal set of each location decreases.

Finally, we conduct scalability tests and study the performance of EB under different dataset cardinalities. Fig. 13 shows the performance of EB as a function of $|D_A|$ (=$|D_B|$). As expected, the cost increases with the cardinality of datasets. EB is designed to evaluate only the records with very promising chances to be part of the result, and hence, it scales well and solves M*k*OLS efficiently even for the case when $|D_A| = |D_B| = 4M$. Note that, we have also tested both NI and BL. However, their performance (e.g., hours for the case when $|D_A| = |D_B| = 4M$) is not comparable with EB, and thus omitted.

6.3. Results on M*k*OLS$_{MR}$ search

The third set of experiments verifies the performance of EBM algorithm designed for the M*k*OLS$_{MR}$ query under different number n of constrained regions, with n changed from one to five. As depicted in Fig. 14, the query cost and the *compdists*

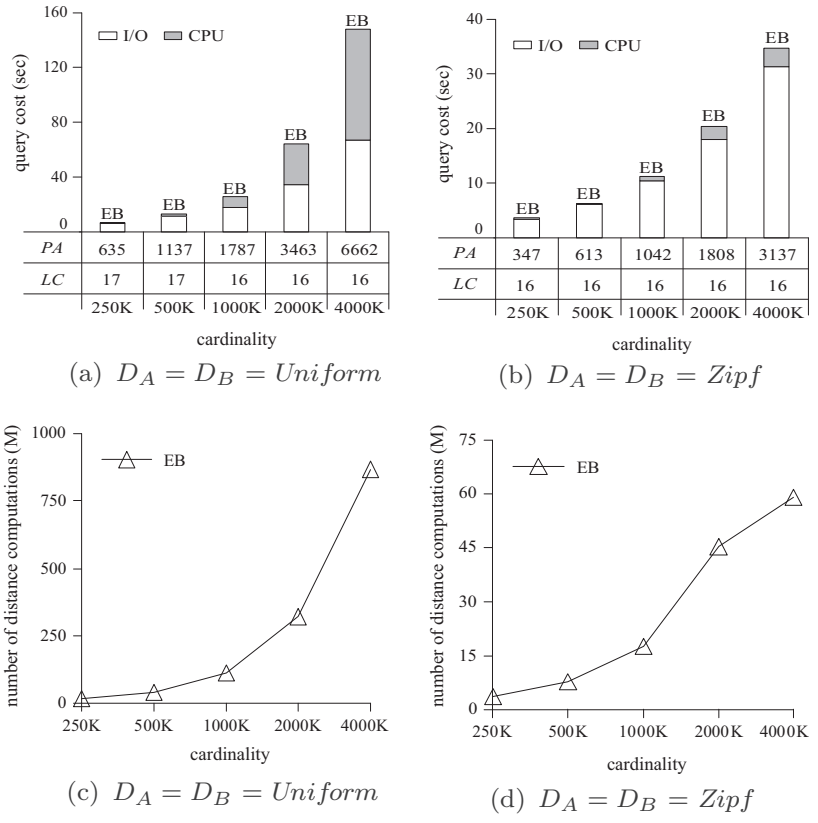


Fig. 13. M*k*OLS performance vs. cardinality.

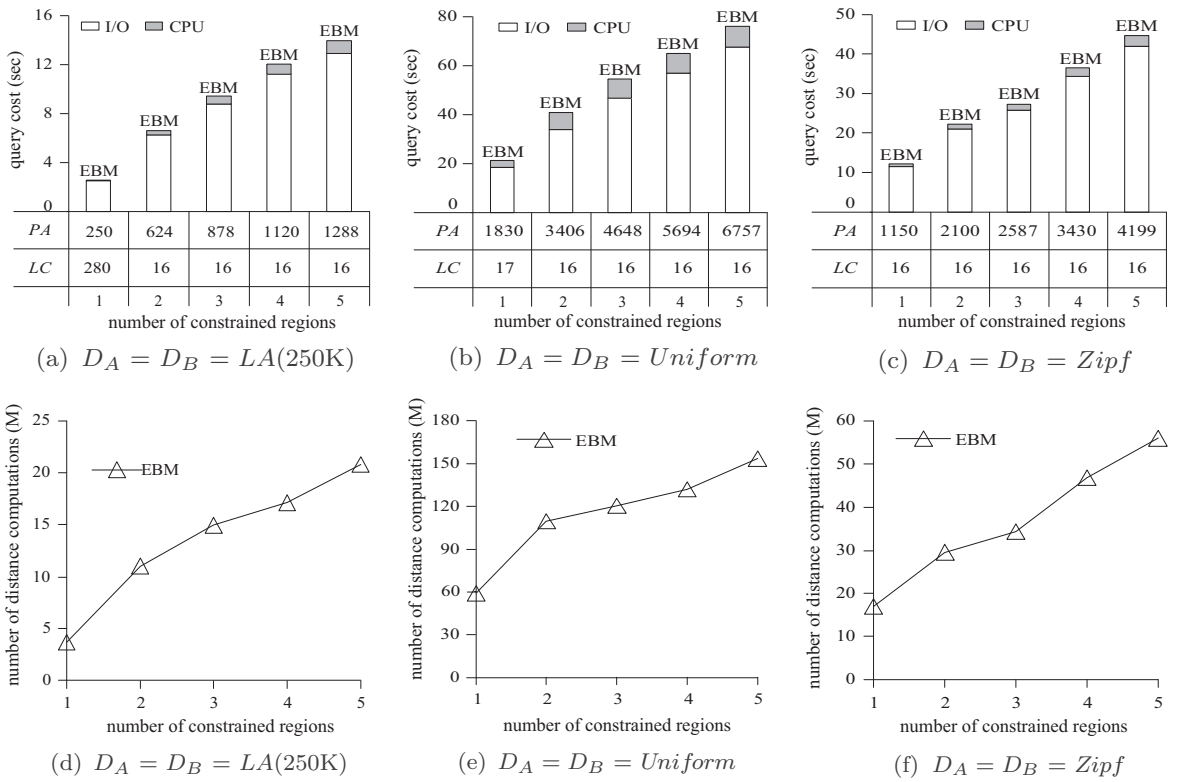


Fig. 14. M*k*OLS$_{MR}$ performance vs. number *n* of constrained regions.

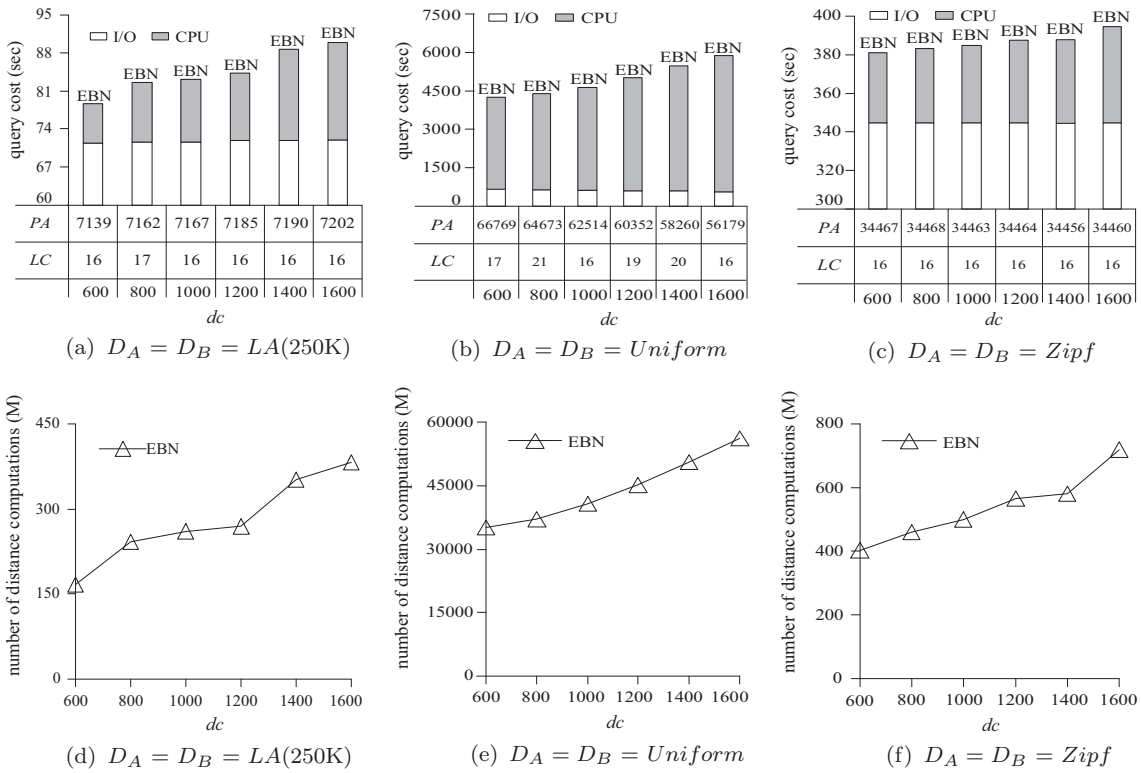


Fig. 15. M*k*OLS$_{NR}$ performance vs. d_c.

increase with the number of *R*. The reason behind is that, with the growth of the number of *R*, the search space becomes larger, leading to more qualified objects and more qualified locations.

6.4. Results on M*k*OLS$_{NR}$ *search*

The last set of experiments evaluates the efficiency of EBN in answering M*k*OLS$_{NR}$ retrieval. According to the definition of the query, it does not consider any constrained region, i.e., the number of regions is 0. We fix *k*, cardinality $|D_A|$, dimensionality to their default values (i.e., 16, 1 M, 2, respectively), and vary d_c from 600 to 1600. Fig. 15 plots the performance of EBN under real and synthetic datasets. The total query time increases with d_c, because more object-location pairs need to be precisely evaluated.

In particular, for *Uniform* datasets, EBN is CPU-bounded instead of I/O-bounded. The reason is that (i) the pruning rules does not work well without any constrained region and (ii) many location entries share an equivalent estimated optimality because of the uniform distribution. It is worth noting that, the CPU cost of EB is small on *Uniform* datasets when there are constrained regions, as reported in previous two sets of experiments. This is because the border of *R* makes the estimated optimality of different locations diverse from each other.

7. Conclusions

This paper, for the first time, identifies and studies the problem of *metric k-optimal-location-selection* (M*k*OLS) *search*, which supports the *k*OLS query in a generic metric space. M*k*OLS retrieval is not only interesting from a research point of view, but also useful in many decision support applications. We propose an efficient algorithm called EB that does not require the detailed representations of the objects, and is applicable as long as the similarity between two objects can be evaluated and meanwhile satisfies the triangle inequality. Our solution is based on metric index structures (i.e., M-trees and COM-trees), employs several pruning rules, and makes use of the reuse and optimality score estimation techniques. Extensive experiments with both real and synthetic datasets demonstrate the effectiveness of the proposed pruning rules and the performance of the proposed algorithms under various problem settings. In addition, we extend our techniques to tackle two interesting and useful M*k*OLS query variants, i.e., M*k*OLS$_{MR}$ and M*k*OLS$_{NR}$ queries.

In the future, we intend to further improve the efficiency of our proposed algorithms by devising better pruning rule(s) and optimality score estimation. For instance, we would like to derive a lower bound of the optimality score to develop effective pruning rules. Also, we plan to explore the applicability of our proposed algorithms in other location facility problems.

Acknowledgements

This work was supported in part by NSFC Grant No. 61379033, the National Key Basic Research and Development Program (i.e., 973 Program) No. 2015CB352502, the Cyber Innovation Joint Research Center of Zhejiang University, and the Key Project of Zhejiang University Excellent Young Teacher Fund (Zijin Plan).

References

[1] E. Achtert, C. Bohm, H.-P. Krieger, P. Kunath, A. Pryakhin, M. Renz, Efficient reverse *k*-nearest neighbor search in arbitrary metric spaces, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2006, pp. 515–526.
[2] E. Achtert, H.P. Kriegel, P. Kroger, M. Renz, A. Zufle, Reverse *k*-nearest neighbor search in dynamic and general metric databases, in: Proceedings of the 12nd International Conference on Extending Database Technology (EDBT), 2009, pp. 886–897.
[3] L.G. Ares, N.R. Brisaboa, A.O. Pereira, O. Pedreira, Efficient similarity search in metric spaces with cluster reduction, in: Proceedings of the 5th International Conference on Similarity Search and Applications (SISAP), 2012, pp. 70–84.
[4] S. Brin, Near neighbor search in large metric spaces, in: Proceedings of the 21st International Conference on Very Large databases (VLDB), 1995, pp. 574–584.
[5] S. Cabello, J. Miguel, D.S. Langerman, C. Seara, I. Ventura, Reverse facility location problems, in: Proceedings of the 17th Canadian Conference on Computational Geometry (CCCG), 2005, pp. 68–71.
[6] E. Chavez, G. Navarro, R. Baeza-Yates, J. Marroquin, Searching in metric spaces, ACM Comput. Surv. 33 (3) (2001) 273–322.
[7] L. Chen, X. Lian, Efficient processing of metric skyline queries, IEEE Trans. Knowl. Data Eng. 21 (3) (2009) 351–365.
[8] Z. Chen, Y. Liu, R.C.W. Wong, J. Xiong, G. Mai, C. Long, Efficient algorithms for optimal location queries in road networks, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2014, pp. 123–134.
[9] P. Ciaccia, M. Patella, P. Zezula, M-tree: An efficient access method for similarity search in metric spaces, in: Proceedings of the 23rd International Conference on Very Large databases (VLDB), 1997, pp. 426–435.
[10] K.L. Clarkson, Nearest neighbor queries in metric spaces, Discr. Comput. Geom. 22 (1) (1999) 63–93.
[11] A. Didandeh, B.S. Bigham, M. Khosravian, F.B. Moghaddam, Using Voronoi diagrams to solve a hybrid facility location problem with attentive facilities, Inform. Sci. 234 (2013) 203–216.
[12] C. Doulkeridis, A. Vlachou, Y. Kotidis, M. Vazirgiannis, Peer-to-peer similarity search in metric spaces, in: Proceedings of the 33rd International Conference on Very Large databases (VLDB), 2007, pp. 986–997.
[13] Y. Du, D. Zhang, T. Xia, The optimal-location query, in: Proceedings of the 6th International Symposium on Spatial and Temporal Databases (SSTD), 2005, pp. 163–180.
[14] M. Fort, J.A. Sellarès, Solving the *k*-influence region problem with the GPU, Inform. Sci. 269 (2014) 255–269.
[15] D. Fuhry, R. Jin, D. Zhang, Efficient skyline computation in metric space, in: Proceedings of the 12nd International Conference on Extending Database Technology (EDBT), 2009, pp. 1042–1051.
[16] Y. Gao, B. Zheng, G. Chen, Q. Li, Optimal-location-selection query processing in spatial databases, IEEE Trans. Knowl. Data Eng. 21 (8) (2009) 1162–1177.
[17] G.R. Hjaltason, H. Samet, Index-driven similarity search in metric spaces, ACM Trans. Database Syst. 28 (4) (2003) 517–580.
[18] J. Huang, Z. Wen, J. Qi, R. Zhang, J. Chen, Z. He, Top-*k* most influential location selection, in: Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM), 2011, pp. 2377–2380.
[19] E.H. Jacox, H. Samet, Metric space similarity joins, ACM Trans. Database Syst. 33 (2) (2008) 7:1–7:38.
[20] F. Korn, S. Muthukrishnan, Influence sets based on reverse nearest neighbor queries, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2000, pp. 201–212.
[21] H. Kurasawa, A. Takasu, J. Adachi, Finding the *k*-closest pairs in metric spaces, in: Proceedings of the 1st International Workshop on New Trends in Similarity Search, 2011, pp. 8–13.
[22] J. Liu, H. Chen, K. Furuse, H. Kitagawa, An efficient algorithm for reverse furthest neighbors query with metric index, in: Proceedings of the 21st International Conference on Database and Expert Systems Applications (DEXA), 2010, pp. 437–451.
[23] K. Mouratidis, D. Papadias, S. Papadimitriou, Tree-based partition querying: a methodology for computing medoids in large spatial datasets, VLDB J. 17 (4) (2008) 923–945.
[24] R. Paredes, N. Reyes, Solving similarity joins and range queries in metric spaces with the list of twin clusters, J. Discr. Algor. 7 (1) (2009) 18–35.
[25] J. Qi, R. Zhang, L. Kulik, D. Lin, Y. Xue, The min-dist location selection query, in: Proceedings of the 28th International Conference on Data engineering (ICDE), 2012, pp. 366–377.
[26] A. Rahmani, S.A. Mirhassani, A hybrid firefly-genetic algorithm for the capacitated facility location problem, Inform. Sci. 283 (2014) 70–78.
[27] Y.N. Silva, S. Pearson, Exploiting database similarity joins for metric spaces, Proc. VLDB Endowment (PVLDB) 5 (12) (2012) 1922–1925.
[28] T. Skopal, J. Pokorny, V. Snosel, PM-tree: Pivoting metric tree for similarity search in multimedia databases, in: Proceedings of the 8th East-European Conference on Advances in Databases and Information Systems (ADBIS), 2004, pp. 99–114.
[29] Y. Tao, M.L. Yiu, N. Mamoulis, Reverse nearest neighbor search in metric spaces, IEEE Trans. Knowl. Data Eng. 18 (9) (2006) 1239–1252.
[30] E.S. Tellez, E. Chavez, K. Figueroa, Polyphasic metric index: Reaching the practical limits of proximity searching, in: Proceedings of the 5th International Conference on Similarity Search and Applications (SISAP), 2012, pp. 54–69.
[31] E. Tiakas, G. Valkanas, A.N. Papadopoulos, Y. Manolopoulos, D. Gunopulos, Metric-based top-*k* dominating queries, in: Proceedings of the 17th International Conference on Extending Database Technology (EDBT), 2014, pp. 415–426.
[32] A. Vlachou, C. Doulkeridis, Y. Kotidis, Peer-to-peer similarity search based on m-tree indexing, in: Proceedings of the 15th International Conference on Database Systems for Advanced Applications (DASFAA), 2010, pp. 269–275.
[33] C. Wang, L. Deng, G. Zhou, M. Jiang, A global optimization algorithm for target set selection problems, Inform. Sci. 267 (2014) 101–118.
[34] T. Xia, D. Zhang, E. Kanoulas, Y. Du, On computing top-*t* most influential spatial sites, in: Proceedings of the 31st International Conference on Very Large databases (VLDB), 2005, pp. 946–957.
[35] X. Xiao, B. Yao, F. Li, Optimal location queries in road network databases, in: Proceedings of the 27th International Conference on Data Engineering (ICDE), 2011, pp. 804–815.
[36] D. Zhang, Y. Du, T. Xia, Y. Tao, Progressive computation of the min-dist optimal-location query, in: Proceedings of the 32nd International Conference on Very Large databases (VLDB), 2006, pp. 643–654.
[37] J. Zhang, W.S. Ku, M.T. Sun, X. Qin, H. Lu, Multi-criteria optimal location query with overlapping Voronoi diagrams, in: Proceedings of the 16th International Conference on Extending Database Technology (EDBT), 2014, pp. 391–402.
[38] J.-D. Zhang, C.-Y. Chow, CoRe: Exploiting the personalized influence of two-dimensional geographic coordinates for location recommendations, Inform. Sci. 293 (2015) 163–181.