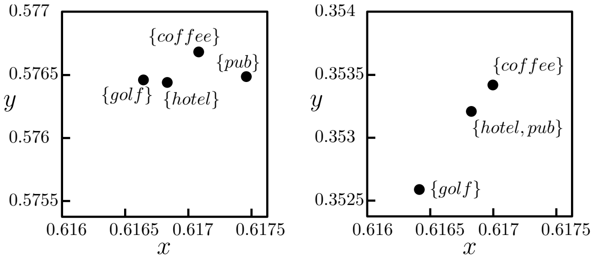
Finding the Minimum Spatial Keyword Cover

Dong-Wan Choi[†] Jian Pei[†] Xuemin Lin[‡]
[†]Simon Fraser University, Burnaby, Canada, {dongwanc, jpei}@sfu.ca
[‡]University of New South Wales, Sydney, Australia, lxue@cse.unsw.edu.au

Abstract*—The existing works on spatial keyword search focus on finding a group of spatial objects covering all the query keywords and minimizing the diameter of the group. However, we observe that such a formulation may not address what users need in some application scenarios. In this paper, we introduce a novel spatial keyword cover problem (SK-COVER for short), which aims to identify the group of spatio-textual objects covering all keywords in a query and minimizing a distance cost function that leads to fewer proximate objects in the answer set. We prove that SK-COVER is not only NP-hard but also does not allow an approximation better than $O(\log m)$ in polynomial time, where m is the number of query keywords. We establish an $O(\log m)$-approximation algorithm, which is asymptotically optimal in terms of the approximability of SK-COVER. Furthermore, we devise effective accessing strategies and pruning rules to improve the overall efficiency and scalability. In addition to our algorithmic results, we empirically show that our approximation algorithm always achieves the best accuracy, and the efficiency of our algorithm is comparable to a state-of-the-art algorithm that is intended for mCK, a problem similar to yet theoretically easier than SK-COVER.



(a) The result of the mCK query (b) The result of SK-COVER

	mCK query	SK-COVER
Cardinality	4	3
Diameter	0.000760	0.000952
$\sum$ distance from hotel	0.001073	0.000952
$\sum$ pairwise distance	0.002713	0.001904
(Cardinality-1) $\times$ Diameter	0.002279	0.001903
$\binom{\text{Cardinality}}{2} \times$ Diameter	0.004558	0.002855

(c) The summary of spatial costs

Fig. 1. Experimental result with the set of query keywords, {*hotel*, *golf*, *coffee*, *pub*}, using the UK dataset

I. INTRODUCTION

You are planning a golf vacation with a few friends, and want to look for a golf course, a hotel, a restaurant and a pub that are close to each other so that you and your friends can hang out as much as possible instead of spending a lot of time on the way. So, you ask a spatial keyword search engine like Google Maps a query "*hotel, golf, restaurant, pub*" and expect that you can get answers, where each answer contains a hotel, a golf course, a restaurant, and a pub so that the distances within a group are as small as possible. It would be nice if multiple facilities are co-located at a place, such as a hotel with a pub.

A series of recent studies [1], [2], [3], [4], [5], [6], [7] proposed efficient algorithms to find a group of objects that cover a given set of query keywords and are proximate to each other in space. While we will review them in Section II, most of them follow the mCK query model [1] and optimize the diameter of the group of objects in space, that is, minimizing the maximum distance between two objects in a group. Can this model really solve the query you have in the above example?

Figure 1 shows the query results on a real dataset about the spatial points of interest (POI) in UK, which is downloaded from http://www.pocketgpsworld.com. We use query {*hotel, golf, restaurant, pub*}. The mCK query returns a group of four POIs covering the four query keywords, as shown in Figure 1(a). While the four POIs are close to each other and are within a small diameter, still you have to go to different places for different activities. Figure 1(b) shows a group of three POIs also covering all the four query keywords. Although those three POIs are within a larger diameter than the group in Figure 1(a), the total distance from the hotel to the other two POIs in the group is indeed smaller than the total distance between the hotel and the other three POIs in the group in Figure 1(a), as detailed in Figure 1(c). Moreover, the sum of pairwise distances in Figure 1(b) is also smaller than that in Figure 1(a). The group in Figure 1(b) is a better choice for you since you can travel a shorter distance among POIs in the group. In addition, the hotel and the pub are co-located.

Motivated by the above observations, in this paper, we propose a new type of spatial keyword queries, namely *spatial keyword cover* (SK-COVER for short). The major difference between SK-COVER and the previous work is that SK-COVER adopts different optimization objectives. Technically, given a set of query keywords, we try to find a group of spatial objects C such that each object is associated with some keywords and all the query keywords are covered, and we try to minimize a distance cost function $f(C)$.

Since from a simple keyword query it is hard to obtain the detailed information about the traversal pattern among the facilities queried, in this paper we consider the following two distance cost functions. In the first case, for a group of objects C, we assume an object serves as the hub, consider the sum of distances from a hub to the other objects in C, and use an upper bound of the sum of distances corresponding to the worst hub in the answer set as the measure of the distance cost. In the second case, we use an upper bound of the sum of all pairwise distances as the measure. In general, other distance functions can be used.

Employing a non-trivial distance cost function other than the diameter makes our SK-COVER problem substantially different from the existing work mainly on the mCK problem [1]. Minimizing the diameter in mCK does not necessarily lead to reducing the number of objects in the answer set. In the distance cost functions considered in our SK-COVER problem, since each object in the answer set incurs a cost, minimizing the cost function tends to reduce the number of objects in the answer set and finding proximate objects simultaneously. As we know, users tend to prefer answers containing fewer objects in spatial keyword searches.

As we will show, the SK-COVER problem is very challenging even we optimize $(|C|-1)\cdot d$ and $\binom{|C|}{2}\cdot d$, respectively, as the upper bounds of the two distance cost functions, where d is the diameter of the set C. Not to mention that SK-COVER is NP-hard as mCK is, it turns out that SK-COVER does not even allow any polynomial time algorithms that can achieve a constant approximation ratio. This is somewhat surprising, since there exist constant-factor approximation algorithms for mCK [8], [7]. Thus, the challenges here are not only how to design an efficient algorithm but also how to achieve the approximation bound as small as possible.

One may think that SK-COVER can be seen as a combination of SET-COVER and mCK. Consequently, it may appear that a simple greedy algorithm can settle our problem, since the well-known greedy algorithm for SET-COVER achieves the optimal approximation bound in polynomial time [9], [10]. Unfortunately, as to be shown, a simple greedy approach adapted from the greedy solution to SET-COVER can produce an unreasonably inaccurate answer whose cost can be $O(m)$ times larger than the optimum in the worst case, where m is the number of query keywords.

In this paper, we propose a fairly accurate solution for settling SK-COVER. The essential idea is that once we fix the diameter part in the cost function, the greedy algorithm for SET-COVER can be used effectively to solve our problem. More specifically, we consider every group of objects whose diameter is no more than the distance between a particular pair of objects in the group. Then, it suffices to choose the group of minimum approximate cost obtained by solving SET-COVER locally with respect to the group. We prove that this strategy guarantees the optimal approximation bound for SK-COVER in polynomial time. To make our algorithm more scalable and practically efficient, we devise effective pruning techniques and advanced accessing methods that can substantially reduce the search space and improve the scalability.

We make the following contributions in this paper. First, we formalize the novel SK-COVER problem of spatial keyword search, which addresses some critical application scenarios where the existing mCK problem is not effective. Second, we prove that SK-COVER is NP-hard and cannot be approximated by a constant factor in polynomial time. Third, we establish a polynomial time approximation algorithm that achieves the asymptotically optimal approximation, and design effective pruning rules and accessing methods to reduce the search space and improve the scalability. Last, we show by experiments that our algorithm gives the most accurate answers all the time. This is consistent with the theoretical quality analysis.

The rest of the paper is organized as follows. Section II surveys the related work. Section III defines the problem, discusses its hardness and the feasibility of extending some existing solutions to our problem. Section IV presents our approximation algorithm as well as the scalable implementation. All our algorithms are experimentally evaluated in Section V. We conclude the paper in Section VI.

II. RELATED WORK

Our SK-COVER problem broadly belongs to the general topic of spatial keyword search, which has received a great deal of attention from the database community in recent years. Earlier works on this topic mainly focus on identifying a single object whose location and textual attribute satisfy a given type of query. A typical problem is finding the closest (or top-k) spatio-textual object(s) containing all the query keywords [11], [12], [13], [14]. Some works on a slightly different version exploit the textual similarity as well as the spatial proximity [15], [16], [17]. All these works are inherently different from SK-COVER in that our goal is to find a group of objects, instead of a single object, that together cover all query keywords. There are also a number of different problems in literature about spatial keyword search. A systematic review of all these problems is far beyond the scope and capacity of this paper. Therefore, in the rest of this section, we focus on only the problems closely related to SK-COVER, namely the *mCK query* problem and the *collective spatial keyword query* (*CoSKQ*) problem. Both problems have similar goals as ours, that is, to find a group of objects covering all keywords and minimizing a given spatial cost function.

A. *mCK Queries*

Given a set of m query keywords, the answer of the mCK query is a group of m objects covering all the keywords and having the smallest diameter, where the diameter of a group is the maximum pairwise distance in the group. This problem was firstly proposed by Zhang *et al.* [1], [2], where an R*-tree based index structure, called the (virtual) bR*-tree, was designed as the underlying structure for their proposed exact algorithms. These works assume that every object is associated with only one keyword. Therefore, an answer group always has exactly m objects. As the mCK query problem has been shown NP-hard [8], [7], the exact algorithm [1], [2] runs in exponential time in the worst case.

To answer mCK queries efficiently, some approximation algorithms have been developed. Fleisher and Xu [8] showed that, when each object has only one keyword, the problem of answering an mCK query can be efficiently and approximately solved by finding the *smallest color-spanning circle*, and provided a $(\frac{2}{\sqrt{3}})$-approximation algorithm using the *farthest color Voronoi diagram* [18]. Recently, Guo *et al.* [7] further relaxed the constraint so that each object can be associated with multiple keywords, and proposed a $(\frac{2}{\sqrt{3}}+\epsilon)$-approximation algorithm.

In addition, Deng *et al.* [6] proposed an exact algorithm for a variant of the mCK query problem, called the *best keyword cover* problem. In this version, each object carries a weight representing a rating score and the objective is to maximize the minimum weight in the group and minimize the diameter of the group through a linear combination.

All the above works on the mCK queries and their variants commonly attempt to minimize the diameter of answer groups. None of them consider the cardinality of answer groups. As shown in Section I, mCK queries cannot fully address what users need in some spatial keyword search scenarios.

B. *Collective Spatial Keyword Queries*

Cao *et al.* [3] introduced the collective spatial keyword queries (CoSKQ), which is an extension of the mCK query problem. The key difference is that a CoSKQ query specifies a particular location. Correspondingly, one additional objective is to minimize the distance between the query location to the result group. The overall objective is to optimize a linear combination of the distance between the query location and the answer group and the diameter of the answer group. They presented both exact algorithms and approximation algorithms along with the proof of NP-hardness of CoSKQ. Long *et al.* [5] further improved the scalability by proposing a *distance owner-driven* approach. Zhang *et al.* [4] studied one variant of CoSKQ, called the *density-based collective spatial keyword* query.

It is obvious that the SK-Cover problem cannot be solved by any of the algorithms for CoSKQ since SK-Cover only specifies a set of query keywords but no query location.

C. Optimal Route Queries

Optimal route queries, also known as trip planning queries [19], are another branch of works related to our motivation. The basic objective of these works is to find the shortest route that covers all the query categories (i.e., keywords). For example, one may want to find the shortest route starting at home and passing through a gas station, a bank, and a post office. Li *et al.* [19] proposed trip planning queries (TPQ), which specify both the starting location and the destination location. Later, Ma *et al.* [20] studied a variant of TPQ without destination locations. Furthermore, Sharifzadeh *et al.* [21] added a total order constraint on the types of locations, and Chen *et al.* [22] and Li *et al.* [23] extended the query by considering partial order constraints. Similar to CoSKQ, all these works cannot be applied to address our problem because they commonly have a starting (or destination) location and optionally an ordering constraint.

III. Problem Definition and Analysis

In this section, we formulate the problem of SK-Cover, examine the hardness and approximability of the problem, and investigate whether the existing methods can be adapted to solve the SK-Cover problem.

A. Problem Formulation

Consider a set O of spatio-textual objects, where each object $o \in O$ is a point in the Euclidean 2D space, and is associated with a set of keywords, denoted by $o.\tau = \{t_1, t_2, \ldots, t_{|o.\tau|}\}$. A keyword query is a set $T = \{t_1, t_2, \ldots, t_{|T|}\}$ of keywords. The *spatial keyword cover problem* (SK-Cover for short) is to find a subset $C \subseteq O$ such that all the query keywords are covered, that is, $T \subseteq \bigcup_{o \in C} o.\tau$, and the following cost function $f(C)$ is minimized.

$$f(C) = (|C| - 1) \cdot \max_{o,o' \in C} dist(o, o'), \qquad (1)$$

where $dist(o, o')$ is the Euclidean distance between o and o'. Here, similar to the methodology adopted in the previous studies on mCK, in the distance cost function we use the diameter of the group as the upper bound of the distance between a pair of objects in the group.

For the ease of presentation, most of this paper will concentrate on the cost function in Equation 1, which corresponds to the maximum hub-based distance mentioned in Section I. Moreover, the distance cost in Equation 1 is also the upper bound of the longest traversal distance among all the points in the answer set. We will discuss how to extend the solution to handle the sum of all pairwise distances in Section IV-D.

B. Hardness and Approximability

We first establish the hardness of SK-Cover as follows.

Theorem 1: The SK-Cover problem is NP-hard.

Proof: We prove the hardness of SK-Cover by reduction from Set-Cover. Let U be a universe set containing all items and $\mathbb{S} = \{S_1, S_2, \ldots\}$ be a collection of sets, where $S_i \subseteq U$. Then, Set-Cover aims to find the smallest collection $\mathbb{X}^* \subseteq \mathbb{S}$ of sets whose union covers U.

To construct the corresponding instance of SK-Cover, we first create $|\mathbb{S}|$ objects in the same location, and associate each object o_i with S_i. Also, we create a dummy object o' having a dummy item (i.e., keyword) $x' \notin U$ at the location whose distance from any o_i is 1, and finally let T be $U \cup \{x'\}$.

If we can find a group $C = \{o', o_1, o_2, \ldots, o_k\}$ of objects, covering T such that $f(C) = (|C| - 1) \cdot dist(o', o_i) = k$ for SK-Cover, then there exists a collection $\mathbb{X} = \{S_1, S_2, \ldots, S_k\}$ of sets covering U of size k for Set-Cover. ■

Since SK-Cover is NP-hard, it is natural to ask how well we can approximate the optimum in SK-Cover.

Theorem 2: There is no polynomial time algorithm that can return an approximate solution to SK-Cover with an approximation bound better than $O(\log |T|)$ unless $P = NP$.

Proof: Due to the proof of Theorem 1, we can see that Set-Cover is a special case of SK-Cover in which only one object with an exclusive keyword lies in a different location and the other objects are restricted to the same location. Also, there is a restriction on T to have all the keywords in the database. This implies that SK-Cover cannot be approximated in a bound better than Set-Cover. The lower bound on approximating Set-Cover in polynomial time is known to be $O(\log n)$ unless $P = NP$ [10], where n is the number of elements to be covered. Thus, SK-Cover does not allow a bound better than $O(\log |T|)$ in polynomial time. ■

C. Adapting the Existing Algorithms

Given the similarity among mCK, Set-Cover and SK-Cover, it is important to check whether the approximation algorithms for those two problems can be straightforwardly extended to settle SK-Cover.

1) Approximating mCK: Since the resulting group of the extended version of the mCK query [7] also covers all the query keywords, can we use a solution for mCK as an approximation for SK-Cover? The following lemma shows that the approximation ratio can be as large as a factor of the input size (i.e., the number of query keywords).

Lemma 1: Let $\hat{C}$ be an optimal group for an instance of the mCK query, and C^* be an optimal group for the corresponding SK-Cover instance. Then, $\frac{f(\hat{C})}{f(C^*)} = O(|T|)$ in the worst case.

Proof: Consider two groups C_1 and C_2, both of which covering T, such that the diameter of C_1 equals that of C_2. Now assume $|C_1| = |T|$ and $|C_2| = 2$. Then, $f(C_1) = \frac{|T|}{2} \cdot f(C_2)$, but the mCK query processing algorithm may return C_1 instead of C_2 because their diameters are tie. ■

Even if we can obtain an optimal answer to an mCK query, the answer is not good with respect to the cost function in SK-Cover.

2) Greedy Approach to Set-Cover*:* For the Set-Cover problem, there is a well-known greedy algorithm, which guarantees an $H(n)$-approximation bound [9], where n is the total number of elements and $H(n) = \sum_{i=1}^{n} \frac{1}{i}$. This bound is asymptotically optimal in terms of the approximability due

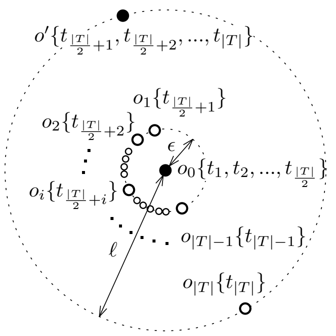

$o'\{t_{\frac{|T|}{2}+1}, t_{\frac{|T|}{2}+2}, ..., t_{|T|}\}$
$o_1\{t_{\frac{|T|}{2}+1}\}$
$o_2\{t_{\frac{|T|}{2}+2}\}$
ϵ
$o_0\{t_1, t_2, ..., t_{\frac{|T|}{2}}\}$
$o_i\{t_{\frac{|T|}{2}+i}\}$
$o_{|T|-1}\{t_{|T|-1}\}$
ℓ
$o_{|T|}\{t_{|T|}\}$

Fig. 2. The worst case example of the greedy approach for SK-COVER

to $H(n) \approx \ln n$ [10]. This fact suggests the possibility of following a greedy scheme to address SK-COVER as well.

The greedy algorithm for SET-COVER picks the set containing the largest number of uncovered elements at each step. Correspondingly, we can choose the most promising object at each step. The issue here is how to measure how promising an object is. Intuitively, a promising object should satisfy the following conditions: (1) it contains as many uncovered keywords as possible; and (2) if there are already some objects selected in the previous steps, it is as close to the selected objects as possible. If we do not consider condition (2), then the algorithm is exactly the same as the greedy algorithm for SET-COVER. Obviously, it does not guarantee any approximation bound for SK-COVER because the diameter of the group of minimum cardinality can be arbitrarily large. Therefore, it is essential to consider both conditions, namely maximizing the *covering power* and minimizing the *incurred distance*.

Unfortunately, even if we carefully design a benefit or pricing function reflecting both the covering power and the incurred distance, such as $price(o) = \frac{\textit{incurred distance by } o}{\textit{covering power of } o}$, the scheme of adding objects one by one in a greedy way still cannot guarantee a non-trivial approximation bound. We intuitively explain this by an example shown in Figure 2, where $T = \{t_1, t_2, \ldots, t_{|T|}\}$, ϵ is close to 0, and $\ell \gg \epsilon$. In this example, the optimal group covering T is $C^* = \{o_0, o'\}$ (plotted in black points) whose diameter is ℓ and therefore $f(C^*) = \ell$. However, the greedy algorithm may output $\hat{C} = \{o_0, o_1, \ldots, o_{|T|}\}$. Here, $f(\hat{C}) = (\ell+\epsilon)(|T|-1) \approx \ell(|T|-1)$. With out loss of generality, we assume that our greedy algorithm selects o_0 as the first object, since both o_0 and o' contain the maximum number of uncovered keywords (i.e.,$\frac{|T|}{2}$) in the first place. Then, for the next promising object, the algorithm may choose o_i instead of o' where $i = 1, 2, \ldots, |T|-1$, since the price of choosing o_i is small enough (due to $\epsilon \ll \ell$) to discard the benefit of choosing o' from covering $\frac{|T|}{2}$ more keywords. Consequently, o' will never be selected until all the keywords of T are covered. In this example, the approximation bound of the simple greedy scheme for SK-COVER can be as large as $O(|T|)$ due to the fact that the incurred distance and the covering power of each object are not correlated at all.

IV. AN ASYMPTOTICALLY OPTIMAL APPROXIMATION ALGORITHM AND SCALABLE TECHNIQUES

In this section, we first develop a polynomial time algorithm that achieves the asymptotically optimum approximability. Then, we discuss several techniques to boost the scalability on large datasets.

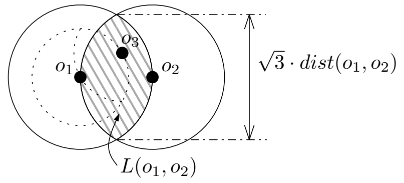


Fig. 3. The lune with respect to o_1 and o_2

ALGORITHM 1: *PolyLune* (O, T)

```
Input: $O$ := a set of objects, $T$ := a set of query keywords
Output: $C$ := a subset of $O$ whose objects together cover $T$
1  $O' \leftarrow$ objects relevant to $T$;    $C \leftarrow \emptyset$;
2  $f_{min} \leftarrow \infty$;
3  foreach $o_1 \in O'$ do
4      foreach $o_2 \in O'$ do
5          $L(o_1, o_2) \leftarrow$ objects inside the lune w.r.t. $o_1$ and $o_2$;
6          if $L(o_1, o_2)$ covers $T$ then
7              $X \leftarrow$ GreedySetCover ($L(o_1, o_2)$, $T$);
8              if $f(X) < f_{min}$ then
9                  $f_{min} \leftarrow f(X)$;
10                 $C \leftarrow X$;
11 return $C$;
```

A. A Polynomial Time Algorithm

We want to devise a new algorithm that guarantees a non-trivial approximation bound.

1) Basic idea: A simple greedy approach fails to achieve a good approximation bound basically because SK-COVER has two aspects to be optimized, namely the diameter and the cardinality of the answer set. Our idea is to fix one aspect for a sub-problem instance and optimize the other aspect with respect to the sub-problem. To obtain the global answer, we can return the best among the results from all sub-problems.

To be more specific, our strategy is to consider a collection $\mathbb{O}$ of subsets of O, in each of which the diameter is fixed to a particular value. We find the best solution among all solutions for the subsets in $\mathbb{O}$. If there is a feasible answer for a given SK-COVER problem instance, then there must exist a subset in $\mathbb{O}$ that covers T, and vice versa.

In order to define such a group of objects whose diameter can be fixed, we need the following notion.

Definition 1 (Lune): Given a pair of objects o_1 and o_2, the *lune* with respect to o_1 and o_2 is the intersection region of the two circles centered at o_1 and o_2, respectively, of radius $dist(o_1, o_2)$.

Figure 3 shows the lune with respect to o_1 and o_2. We use $L(o_1, o_2)$ to denote the set of objects inside the lune with respect to o_1 and o_2. The lune with respect to o_1 and o_2 has the following properties.

Property 1: Every object within a distance up to $dist(o_1, o_2)$ from both o_1 and o_2 must be in $L(o_1, o_2)$.

Object o_3 in Figure 3 is an example for Property 1.

Property 2: For any objects o and o' in $L(o_1, o_2)$, $dist(o, o') \leq \sqrt{3} \cdot dist(o_1, o_2)$.

2) The Algorithm: Algorithm 1 shows the pseudocode of our approximation method *PolyLune*.

First, we filter out the objects not containing any keywords in T. Let O' be the set of remaining objects (Line 1). Denote by f_{min} the minimum cost identified so far, which is initially set to ∞ (Line 2). For the lune with respect to each pair of objects o_1 and o_2 in O', if $L(o_1, o_2)$ covers T, then we perform the greedy algorithm for SET-COVER, namely *GreedySetCover* [9], on $L(o_1, o_2)$ by fixing the diameter to $dist(o_1, o_2)$ (Lines 3-7). The distance cost of the resulting group X, denoted by $f(X)$, is calculated using Equation 1. If this cost is smaller than the current value of f_{min}, then we update the set C and f_{min} accordingly (Lines 8-10). Finally, the best C of the smallest f_{min} value is returned (Line 11).

3) Analysis: The correctness of *PolyLune* is established by the following lemma.

Lemma 2: Given a set of objects O and a keyword query T, there exists a group $C' \subseteq O$ covering T if and only if *PolyLune* returns a non-empty group covering T.

Proof: Consider two objects $o'_1, o'_2 \in C'$ such that $dist(o'_1, o'_2)$ is the diameter of C'. Then, every object in C' is in lune $L(o'_1, o'_2)$ due to Property 1, that is, $C' \subseteq L(o'_1, o'_2)$. This lune must be processed by the algorithm because the algorithm checks every pair of objects in O'. It is easy to see that the other way is also true. ■

Theorem 3 (Complexity): Algorithm *PolyLune* has time complexity $O(|O'|^2 \sum_{o \in O'} |o.\tau| + |O||T| + \sum_{o \in O} |o.\tau|)$.

Proof: First, identifying all objects relevant to T requires $O(|O||T| + \sum_{o \in O} |o.\tau|)$ time by checking for each object $o \in O$ if $o.\tau \cap T \neq \emptyset$. For each pair of objects $o_1, o_2 \in O'$, we retrieve all objects in $L(o_1, o_2)$, which entails $O(|O'|)$ time, and then *GreedySetCover* is performed. Using the method invented by Cormode *et al.* [24], *GreedySetCover* can be done in time linear to the total number of items in the collection, and therefore it entails $O(\sum_{o \in L(o_1, o_2)} |o.\tau|)$ time for each pair. Since $\sum_{o \in L(o_1, o_2)} |o.\tau|$ is up to $\sum_{o \in O'} |o.\tau|$ that dominates $O(|O'|)$, the total cost of processing all pairs is $O(|O'|^2 \sum_{o \in O'} |o.\tau|)$. The theorem is proved.

Please note that the cost $O(|O'|^2 \sum_{o \in O'} |o.\tau|)$ likely dominates the entire complexity in practice because it is most likely to be higher than $O(|O||T| + \sum_{o \in O} |o.\tau|)$. ■

Now let us examine the approximation bound guaranteed by the *PolyLune* algorithm.

Lemma 3: The *PolyLune* algorithm returns an $O(\log |T|)$-approximate answer for SK-COVER.

Proof: Let C^* be the optimal group, and $\hat{C}$ be the group returned from *PolyLune*. Then, we need to prove:

$$f(\hat{C}) \leq O(\log |T|) \cdot f(C^*).$$

Let r^* be the diameter of C^*. Then, $f(C^*) = r^* \cdot (|C^*| - 1)$. Also, there must be two objects $o^*_1, o^*_2 \in C^*$ such that $dist(o^*_1, o^*_2) = r^*$, implying that there must be the corresponding lune with respect to o^*_1 and o^*_2 such that $C^* \subseteq L(o^*_1, o^*_2)$ by Property 1. Since the algorithm investigates every pair of objects, it also performs *GreedySetCover* on $L(o^*_1, o^*_2)$ and returns the corresponding set $X^* \subseteq L(o^*_1, o^*_2)$. Due to the theorem on the approximation bound of *GreedySetCover* [9], we have $|X^*| \leq (\ln |T| + 1) \cdot |C^*|$.

Now let us consider $L(\hat{o_1}, \hat{o_2})$, where $\hat{C}$ comes from. According to the *PolyLune* algorithm, it holds that $f(\hat{C}) \leq f(X^*)$. Also, by Property 2, it holds that $f(X^*) \leq \sqrt{3} \cdot r^* \cdot (|X^*| - 1)$. Combining all together, we obtain

$$\begin{aligned} f(\hat{C}) &\leq f(X^*) \leq \sqrt{3} \cdot r^* \cdot (|X^*| - 1) \\ &\leq \sqrt{3} \cdot (\ln |T| + 1) \cdot \frac{|C^*|}{|C^*| - 1} \cdot r^* \cdot (|C^*| - 1) \\ &= O(\log |T|) \cdot f(C^*) \end{aligned}$$

■

The following theorem states one of the main results of this paper, which follows directly from Theorem 2 and Lemma 3.

Theorem 4: PolyLune is asymptotically optimal in terms of the approximability of SK-COVER.

B. Enhancing the Scalability

Although *PolyLune* runs in polynomial time, it is still costly on a large spatio-textual dataset. Now let us examine the steps in Algorithm 1 one by one, and explore how to implement the steps efficiently.

1) Initialization: In the first step, *PolyLune* computes O', the set of objects relevant to T. To this end, we use inverted lists as our underlying index on the entire dataset O rather than a big spatial index augmented with textual information. This decision is based on the fact that, for this type of spatial keyword queries not involving a query location, using inverted lists outperforms a big spatial index [2]. This is probably because the effect of spatial pruning is much weaker than that of textual pruning.

During this step, we also build two structures on O' that will be used later. The first one is a virtual bR*-tree [2]. The other one is a hash table, denoted by $\mathcal{HT}$, which maps a cardinality value w to the corresponding set of all objects that have w keywords, that is, $\mathcal{HT}[w]$ returns a set of objects having w keywords.

While retrieving each object relevant to T from the inverted lists corresponding to the keywords in T, we do the following. First, for each object o such that $o.\tau \cap T \neq \emptyset$, that is, o is relevant to T, we make a copy of o, discard all irrelevant keywords in $o.\tau$, that is, those keywords in $o.\tau - T$, and add the copy of o into O'. After this step, for every object $o \in O'$ only the keywords relevant to T are considered and the irrelevant keywords are discarded. Thus, for any $o \in O'$, $|o.\tau| = |o.\tau \cap T|$. Hereafter, for the sake of simplicity, we overload the notion $o.\tau$ to denote $o.\tau \cap T$.

Second, we insert the copy $o \in O'$ into the virtual bR*-tree as in [2]. Finally, we insert the copy o into the $|o.\tau|$-th slot of the hash table (i.e., $\mathcal{HT}[|o.\tau|]$) that is the set of objects having exactly $|o.\tau|$ keywords relevant to T.

Through this process, we can substantially reduce the size of the virtual bR*-tree and the cost of processing each $o.\tau$. Furthermore, the task of removing all irrelevant keywords does not require scanning all the keywords originally residing in $o.\tau$. Instead, we first set $o.\tau = \emptyset$ for every $o \in O'$. Then, for each object o_t in the inverted list of a keyword $t \in T$, we add t to $o_t.\tau \cap T$, which requires only $|o.\tau \cap T|$ insertions for each $o \in O'$.

2) Pruning Strategies: PolyLune needs to check the lune with respect to every pair of objects in O', which is the major cost in the algorithm. We suggest two main pruning strategies, namely *spatial pruning* and *cardinality pruning* to speed up this step.

a) Spatial pruning: We observe that, for an object $o_i \in O'$, not every other object in O' needs to be considered as a partner of o_i for checking the corresponding lune. Specifically, if we know the lower bound on the cardinality of any feasible group covering T, then we can limit the search space for each object as stated in the following result.

Lemma 4: Let f_{min} be the minimum cost obtained so far in *PolyLune*, and k_{LB} be a lower bound on the cardinality of any group covering T. For any $o_1, o_2 \in O'$, if $dist(o_1, o_2) \cdot$

$(k_{LB}-1) \geq f_{min}$, then the lune $L(o_1, o_2)$ cannot have a group whose cost is smaller than f_{min}.

Thus, for each $o_i \in O'$, it suffices to consider only the other objects whose distances from o_i are smaller than $\frac{f_{min}}{k_{LB}-1}$. The smaller f_{min} obtained during the process, the smaller search region for each of the objects to be checked.

Now the remaining question is how to get k_{LB} as large as possible. To obtain a reasonably tight k_{LB}, we perform *GreedySetCover* on O' regardless of the diameter aspect. Let $\hat{x}$ be the cardinality of the resulting group. Then, we set k_{LB} to $|\hat{x}|$. Note that this does not invalidate Lemma 3, since $\hat{x}$ is at most $(\ln |T| + 1)$ times larger than the optimum [9].

b) Cardinality pruning: This pruning technique makes good use of the accessing order of the objects in O'. Basically, we should traverse the objects in the descending order of their textual cardinalities (i.e., $|o.\tau|$) using our hash table $\mathcal{HT}$ built in the initialization step. More specifically, we use an integer variable $w \in [1, \max_{o \in O'} |o.\tau|]$ that stores a cardinality value. Then, we access objects in O' as follows. First, we set w to $\max_{o \in O'} |o.\tau|$. Then, for each $o \in \mathcal{HT}[w]$, we check every lune with respect to o and $o' \in O'$ such that $dist(o, o') < f_{min}/k_{LB}$ according to Lemma 4. If $w > 1$, then we decrease w by one and repeat the same process.

One nice property of this accessing scheme is that we can terminate the entire process earlier if we know the lower bound on the diameter of any feasible group covering T. The correctness of this is guaranteed by the following result.

Lemma 5: Let r_{LB} be the lower bound on the diameter of any group covering T. Then, the algorithm *PolyLune* can terminate once $\lceil \frac{|T|}{w} - 1 \rceil \cdot r_{LB} \geq f_{min}$.

Proof: First of all, since w is the current upper bound of the cardinality value, each of the unprocessed objects that have not been checked must have w or less keywords. In order to cover T, at least $\lceil \frac{|T|}{w} \rceil$ objects are required. Therefore, the cost of using some unprocessed objects is at least $\lceil \frac{|T|}{w} - 1 \rceil \cdot r_{LB}$. The claim follows immediately. ■

How to get a good estimation of r_{LB}? Actually, the maximum r_{LB} is the exact answer of the corresponding mCK query. For the efficiency, we use the greedy approximation algorithm in [7], which returns a 2-approximation. Therefore, if $\hat{r}$ is returned from the greedy mCK query processing algorithm, then r_{LB} is set to $\hat{r}/2$.

3) Reducing the Cost of Processing Each Lune: Our pruning rules can reduce the search space. However, for the objects that are not pruned, we still need to retrieve them in lunes, that is, to test for each of the lunes defined on the remaining objects whether the lune covers T. A naïve implementation is to perform a 2D range search on the virtual bR*-tree on O' with the area of each lune to identify the objects in the lune, and check if those objects together cover T. The cost of processing the lune with respect to a pair $o_1, o_2 \in O'$ is $O(\sqrt{|O'|} + \sum_{o \in L(o_1,o_2)} |o.\tau|)$, where the former part is due to a range search [25] and the latter part is due to scanning all the objects in the lune for the covering test. This is a large overhead, since it applies to every pair of objects not pruned.

In order to make this operation more efficient, we observe the following. For objects $o_1, o_2 \in O'$, the lune with respect to o_1 and o_2 belongs to the circle of radius $dist(o_1, o_2)$ centered at o_1. Moreover, for $o_1, o_2, o_3 \in O'$ such that $dist(o_1, o_3) \leq dist(o_1, o_2)$, the lune with respect to o_1 and o_3 also belongs to the circle of radius $dist(o_1, o_2)$ centered at o_1 (see Figure 3).

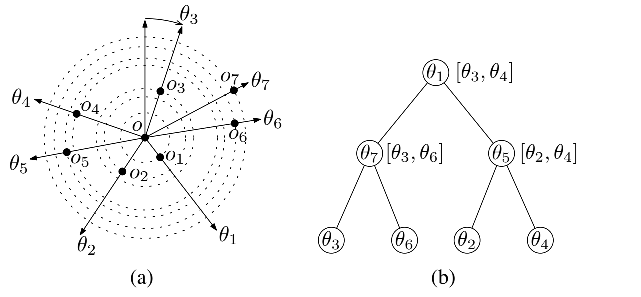


Fig. 4. The polar coordinate system with respect to o and its corresponding polar-tree

Accordingly, if a circle centered at o_1 does not cover T, we can prune all lunes inside the circle, such as the lune with respect to o_1 and o_3 such that $dist(o_1, o_3)$ is up to the the radius of the circle, because those lunes cannot cover T, either.

Moreover, the lune with respect to o_1 and o_3 can help to process the lune with respect to o_1 and o_2. Thus, for each $o \in O'$, if we process the other objects as a partner of o in the ascending order of the distance from o, then all objects previously processed can be *reused* for processing the lune with respect to o and the partner of o being considered. To this end, we first design a structure for each $o \in O'$, which maintains the set of *nearest neighbors* (*NNs* for short) of o, called the *polar-tree*.

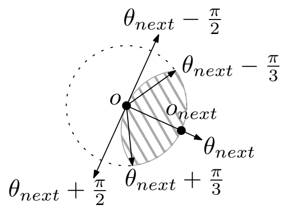
a) Polar-tree: To define the polar-tree, let us first consider a polar coordinate system with respect to $o \in O'$ and NNs of o, in which each NN of o is represented by an angle from a particular direction and a distance from o. In our polar coordinate system, we use clockwise angles where the 12 o'clock position is 0. Then, the polar-tree with respect to o is defined as follows.

Definition 2 (Polar-tree): For $o \in O'$, the polar-tree of o, denoted by $\mathcal{PT}_o$, is a binary search tree on the angles of NNs of o, where each node $u \in \mathcal{PT}_o$ is augmented with the following information.

- $\theta(u)$: its key, i.e., an angle;
- $o(u)$: the corresponding object, i.e., one of NNs of o whose angle is $\theta(u)$;
- $d(u)$: the distance from o to $o(u)$, i.e., $dist(o, o(u))$;
- $I(u)$: an angle range $[\theta_{min}, \theta_{max}]$, where θ_{min} and $;\theta_{max}$ are the minimum angle and the maximum angle, respectively, in the subtree of u; and
- $W(u)$: the set of keywords contained by $o(u)$ and all descendants of $o(u)$.

Example 1: Figure 4 shows an example of the polar-tree with respect to o. In Figure 4(a), the i-th NN of o is denoted by o_i, and its corresponding angle is denoted by θ_i. In the clockwise direction, angles are θ_3, θ_7, θ_6, θ_1, θ_2, θ_5, and θ_4 in order. The polar-tree of o is shown in Figure 4(b), where, for each node u, $\theta(u)$ is in the circle depicting u and $I(u)$ is also shown in the right side of the circle for non-leaf nodes.

b) Processing Each Object Using the Polar-tree: Using the polar-tree, we can process each lune as follows. For each $o \in O'$, we incrementally retrieve the NNs of o by performing the nearest neighbor search on the virtual bR*-tree on O' that is built in the initialization step. We insert each retrieved NN of o, denoted by o_{next}, into the polar-tree of o, denoted by $\mathcal{PT}_o$, using its angle as the key. During the insertion, we update $I(u)$ and $W(u)$ accordingly for each node u along the search



$\theta_{next} - \frac{\pi}{2}$
$\theta_{next} - \frac{\pi}{3}$
o
o_{next}
θ_{next}
$\theta_{next} + \frac{\pi}{2}$
$\theta_{next} + \frac{\pi}{3}$

Fig. 5. The angle range for $L(o, o_{next})$

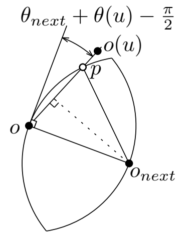


Fig. 6. Lemma 6

path. Moreover, we perform a range search on $\mathcal{PT}_o$ in order to retrieve all the objects residing in the lune with respect to o and o_{next}.

Thanks to the polar-tree, we can concentrate on only the NNs of o that are already processed in order to process the next NN of o being retrieved.

The next question is how to define a range for each o_{next} to retrieve all the objects in $L(o, o_{next})$. Let θ_{next} be the angle of o_{next}. We can obtain all the objects in $L(o, o_{next})$ by fetching every object from $\mathcal{PT}_o$ whose angle is in the range $[\theta_{next} - \frac{\pi}{2},\ \theta_{next} + \frac{\pi}{2}]$. As illustrated in Figure 5, the set of objects returned from this range search may be a superset of $L(o, o_{next})$. Thus, we need to filter out the objects that are not in $L(o, o_{next})$. Instead of exhaustively checking every object in the resulting set, we conduct this refinement task in a piggybacking manner during the range search using the following result.

Lemma 6: For $o \in O'$, let o_{next} be the next NN of o being retrieved of angle θ_{next}, and let $r = dist(o, o_{next})$. For each node u fetched from the range search on $\mathcal{PT}_o$ with range $[\theta_{next} - \frac{\pi}{2},\ \theta_{next} + \frac{\pi}{2}]$, the following statements hold:

1) If $\theta(u) \in [\theta_{next} - \frac{\pi}{3},\ \theta_{next} + \frac{\pi}{3}]$, then $o(u) \in L(o, o_{next})$.
2) If $2r \sin(\theta_{next} + \theta(u) - \frac{\pi}{2}) \geq d(u)$, then $o(u) \in L(o, o_{next})$.

Proof: Statement 1 is apparent as illustrated in Figure 5. To prove Statement 2, we consider one general example shown in Figure 6, where $\theta(u) \in [\theta_{next} - \frac{\pi}{2},\ \theta_{next} + \frac{\pi}{2}]$ but $o(u) \notin L(o, o_{next})$. In order for $o(u)$ to belong to $L(o, o_{next})$, $dist(o, o(u))$, i.e., $d(u)$, must be less than or equal to the length of $\overline{op}$. Because the length of $\overline{o_{next}p}$ is equal to $dist(o, o_{next}) = r$, the triangle poo_{next} is an isosceles triangle. Therefore, the length of $\overline{op}$ is $2 \times r\cos(\angle poo_{next})$, which is equal to $2r\sin(\theta_{next} + \theta(u) - \frac{\pi}{2})$. ■

Using Lemma 6, whenever we encounter a node u whose $\theta(u)$ is within range $[\theta_{next} - \frac{\pi}{2},\ \theta_{next} + \frac{\pi}{2}]$, we can determine whether $o(u)$ is a false positive.

It is worth noting that a range search on the polar-tree is much cheaper than a 2D window range search on the virtual bR*-tree due to the following reasons. First, since the polar-tree is just a 1D binary search tree, a 1D range search on the polar-tree entails only $O(\log n)$ time, which is far less than the cost of a 2D range search on the virtual bR*-tree (or any 2D index structures), which is $O(n^{1/2})$ [25], where n is the total number of elements. Also, each polar-tree with respect to $o \in O'$ maintains only the NNs of o, and hence its size will be much smaller than that of the virtual bR*-tree on O'.

c) Early Test on Covering T: The polar-tree often enables us to check whether a lune covers T (or not) without fetching all the objects in the lune. For this early test, we use the augmented values in each node u, namely $I(u)$ and $W(u)$.

Let o_{next} be the next NN being retrieved, and θ_{next} be its angle based on $o \in O'$. During the insertion of o_{next} into $\mathcal{PT}_o$, we can determine early whether $L(o, o_{next})$ covers T or not by considering the nodes along the search path using the following result.

Lemma 7: If $\{o, o_{next}\}$ do not cover T, then

1) For the root node u of $\mathcal{PT}_o$, if $|W(u) \cup o_{next}.\tau| < |T|$ or $I(u) \cap [\theta_{next} - \frac{\pi}{2}, \theta_{next} + \frac{\pi}{2}] = \emptyset$, then $L(o, o_{next})$ cannot cover T.
2) For any node $u \in \mathcal{PT}_o$, if $I(u) \supseteq [\theta_{next} - \frac{\pi}{2}, \theta_{next} + \frac{\pi}{2}]$ and $|W(u) \cup o_{next}.\tau| < |T|$, then $L(o, o_{next})$ cannot cover T.
3) For any node $u \in \mathcal{PT}_o$, if $I(u) \subseteq [\theta_{next} - \frac{\pi}{3}, \theta_{next} + \frac{\pi}{3}]$ and $|W(u) \cup o_{next}.\tau| = |T|$, then $L(o, o_{next})$ covers T.

Proof: Since each $o \in O'$ has only keywords relevant to T, $|W(u) \cup o_{next}.\tau| < |T|$ if and only if $(W(u) \cup o_{next}.\tau) \subset T$ and hence $W(u) \cup o_{next}.\tau$ does not cover T.

1) In this case, either the entire set of objects in $\mathcal{PT}_o$ together with o_{next} still cannot cover T, or there is no object in $L(o, o_{next})$ other than o and o_{next}.
2) In this case, $I(u) \supseteq [\theta_{next} - \frac{\pi}{2}, \theta_{next} + \frac{\pi}{2}]$ if and only if $(W(u) \cup o_{next}.\tau) \supseteq \bigcup_{o \in L(o, o_{next})} o.\tau$.
3) Similar to the above case, $\bigcup_{o \in L(o, o_{next})} o.\tau \supseteq (W(u) \cup o_{next}.\tau) = T$

■

Note that this early test is performed during the insertion of each NN retrieved into the polar-tree, and hence it does not entail any additional cost. Also, updating both $W(u)$ and $I(u)$ for each node along the insertion path can obviously be done in constant time. Similar to the virtual bR*-tree, a keyword set insertion can be done in constant time by a bitmap structure.

C. The Scalable Algorithm

Now we present a scalable version of our approximation algorithm, called *ScaleLune*, in Algorithm 2, which follows the overall flow of *PolyLune* in Algorithm 1 yet employs all the techniques explained in Section IV-B.

We start with copying all the objects relevant to T into O' from the inverted lists on O and discarding irrelevant keywords from each of relevant objects. In the meantime, we gradually build the virtual bR*-tree on O' and a hash table on O' as explained in Section IV-B1 (Lines 1-3). We also compute k_{LB} and r_{LB}, which are used in our pruning scheme. As explained in Section IV-B2, k_{LB} can be obtained from *GreedySetCover* on O', and r_{LB} is returned from the greedy algorithm for answering the mCK query [7], namely *GreedyMCK* (Lines 4-5). Also, we initialize the variables f_{min} and C (Line 6).

The next step is to process every lune with respect to a pair of objects in O' (Lines 7-25). According to our cardinality pruning scheme presented in Section IV-B2, we traverse every object in O' in the descending order of $|o.\tau|$ (Line 7). To this end, we employ our hash table, denoted by $\mathcal{HT}_{O'}$, which maps a cardinality value w to a set of objects having exactly w keywords, namely $\mathcal{HT}_{O'}[w]$ (Line 8). For each $o \in \mathcal{HT}_{O'}[w]$, we maintain the polar-tree with respect to o on the NNs of o, denoted by $\mathcal{PT}_o$ (Lines 9 and 14). By means of the virtual bR*-tree on O, denoted by $b\mathcal{RT}_{O'}$, we incrementally retrieve each o_{next}, namely *GetNextNN* (Lines 10 and 25).

After o_{next} is retrieved, we insert it into the polar-tree being maintained for o. As explained in Section IV-B3, during each insertion of o_{next}, we also check whether the lune with

ALGORITHM 2: *ScaleLune* (O, T)

```
Input: $O$ := a set of objects, $T$ := a set of query keywords
Output: $C$ := a subset of $O$ whose objects together cover $T$
1  $O' \leftarrow$ objects relevant to $T$ from inverted lists on $O$ while
   removing irrelevant keywords from $o.\tau$ for each $o \in O'$;
2  $b\mathcal{RT}_{O'} \leftarrow$ build the virtual bR$^*$-tree on $O'$;
3  $\mathcal{HT}_{O'} \leftarrow$ build a hash table on $O'$;
4  $C_{sc} \leftarrow$ GreedySetCover ($O'$, $T$);  $k_{LB} \leftarrow \max\{2, |C_{sc}|\}$;
5  $C_{mck} \leftarrow$GreedyMCK ($O'$, $T$);  $r_{LB} \leftarrow \frac{1}{2} \cdot$ diameter of $C_{mck}$;
6  $f_{min} \leftarrow \min\{f(C_{sc}), f(C_{mck})\}$;   $C \leftarrow \emptyset$;
7  for $w \leftarrow \max_{o \in O'} |o.\tau|$ to 1 do
8      foreach $o \in \mathcal{HT}_{O'}[w]$ do
9          $\mathcal{PT}_o \leftarrow$ create the polar-tree w.r.t. $o$;
10         $o_{next} \leftarrow b\mathcal{RT}_{O'}$.GetNextNN($o$);
11         while $dist(o, o_{next}) < \frac{f_{min}}{k_{LB}-1}$ do
12             if $\lceil \frac{|T|}{w} - 1 \rceil \cdot r_{LB} \geq f_{min}$ then
13                 return $C$;
14             $isCovering \leftarrow \mathcal{PT}_o$.Insert($o_{next}$);
15             if $|o_{next}.\tau| > w$ then continue ;
16             if $isCovering \neq$ false then
17                 $L(o, o_{next}) \leftarrow \mathcal{PT}_o$.RangeSearch($o_{next}$);
18                 if $isCovering =$ uncertain then
19                     $isCovering \leftarrow$ check if $L(o, o_{next})$ covers $T$;
20                 if $isCovering =$ true then
21                     $X \leftarrow$ GreedySetCover ($L(o_1, o_2)$, $T$);
22                     if $f(X) < f_{min}$ then
23                         $f_{min} \leftarrow f(X)$;
24                         $C \leftarrow X$;
25             $o_{next} \leftarrow b\mathcal{RT}_{O'}$.GetNextNN($o$);
26 return $C$;
```

respect to o and o_{next} covers T or not, and the answer is returned from the insertion function and stored in variable $isCovering$ (Line 14). Also, according to our cardinality pruning scheme, we skip o_{next} if $|o_{next}.\tau|$ is larger than the current cardinality value being considered (i.e., w) (Line 15). The variable $isCovering$ can have one of the following values: ***true***, ***false***, and ***uncertain***. Only if $isCovering$ is not ***false***, we proceed the next step (Line 16). In both cases of ***true*** and ***uncertain***, it is essential to load all the objects in the lune into $L(o, o_{next})$. This retrieval can be done using the polar-tree as specified in Section IV-B3 (Line 17). After we establish $L(o, o_{next})$, we have to check if $L(o, o_{next})$ covers T when $isCovering =$ ***uncertain*** (Lines 18-19). If $L(o, o_{next})$ covers T, we do the same process with $L(o, o_{next})$ as *PolyLune* to get an approximate local solution with respect to $L(o, o_{next})$ (Lines 20-24). For each $o \in \mathcal{HT}_{O'}[w]$, the incremental NN retrieval of o is terminated when the next NN being retrieved is not closer to o than $f_{min}/(k_{LB}-1)$ according to our spatial pruning scheme (Lines 12-13).

Finally, at any time during the entire process, when the cardinality pruning condition in Lemma 5 is satisfied, we can immediately stop and return C (Lines 12-13).

1) Theoretical analysis: The correctness and the approximation bound of *PolyLune* (i.e., Lemmas 2 and 3, and Theorem 4) are still valid for *ScaleLune*. The cost of *ScaleLune* is analyzed as follows.

a) Initialization: (Lines 1-6) Scanning all inverted lists relevant to T entails $O(\sum_{t \in T} |O_t|)$ time, where O_t is the set of all objects containing keyword t, due to merging all the lists relevant to T. The cost of removing all irrelevant keywords is $O(\sum_{o \in O'} |o.\tau \cap T|)$ as clarified in Section IV-B1, which is in fact identical to $O(\sum_{t \in T} |O_t|)$. Also, the cost of building the virtual bR*-tree and that of building the hash table are $O(|O'| \log |O'|)$ and $O(|O'|)$, respectively. *GreedySetCover* on O' entails $O(\sum_{o \in O'} |o.\tau \cap T|)$ time using the fastest SET-COVER algorithm by Cormode *et al.* [24]. Finally, *GreedyMCK* requires $O(|T| \cdot \min_{t \in T} |O_t| \cdot g(|O'|))$ time [7], where $g(|O'|)$ is the cost of each retrieval of NNs using keywords. Hence, the total cost of the initialization can be summarized as

$$O\left(\sum_{t \in T} |O_t| + |O'| \log |O'| + g(|O'|)|T| \min_{t \in T} |O_t|\right) \qquad (2)$$

b) The Main Part: (Lines 7-25) Let $O'' \subseteq O'$ be the set of objects that cannot be pruned by our cardinality pruning scheme. Then, the entire cost of the main part is $|O''|$ times the cost of processing each $o \in \mathcal{HT}_{O'}[w]$. For each $o \in \mathcal{HT}_{O'}[w]$, let N_o be the set of NNs of o to be retrieved. Then, N_o contains all NNs of o that are not pruned by our spatial pruning scheme. The cost can be represented as $O(y(|O'|) \sum_{o \in O''} |N_o| + \sum_{o \in O''} h(o)|N_o|)$, where $y(|O'|)$ is the cost of each retrieval of NNs that is theoretically achieved as $O(\log |O'|)$ [26], and $h(o)$ is the total cost of additional tasks for each NN of o. For each NN of o, say o_{next}, we need to insert o_{next} into the polar-tree, perform a range search on the polar-tree, and occasionally invoke *GreedySetCover* on $L(o, o_{next})$. The cost of an insertion of the polar-tree is $O(\log |N_o|)$ that is also the cost of a range search because the polar-tree is just a binary search tree on 1D values (i.e., angles). To preserve the balance, the polar-tree is implemented as a *red-black tree*.*GreedySetCover* on each lune is performed only when the lune covers T. Let N'_o be the set of NNs of o such that the lune with respect to o and each $o_{next} \in N'_o$ covers T. Then, the cost of processing *GreedySetCover* is $O(\sum_{o' \in N'_o} |o'.\tau \cap T|)$. Since $O(\log |N_o|)$ is dominated by $O(y(|O'|)) \approx O(\log |O'|)$, the cost of the main part can finally be represented as

$$O\left(\log |O'| \sum_{o \in O''} |N_o| + \sum_{o \in O''} \sum_{o' \in N'_o} |o'.\tau \cap T|\right) \qquad (3)$$

The cost in Equation 3 can be regarded as the total cost of *ScaleLune* in practice, since it is most likely to dominate the cost in Equation 2. This cost is much lower than the cost in Theorem 3 in the sense that, in practice, more often $|N'_o| < |N_o| < |O''| \ll |O'|$. Not only that, theoretically *ScaleLune* is an *output-sensitive* algorithm since its entire cost highly depends on how many combinations of objects cover T. Actually, when there are not many groups of objects covering T, then $|N'_o| \ll |N_o|$, which means the cost in Equation 3 is close to $O(\log |O'| \sum_{o \in O''} |N_o|)$.

D. Extension to Sum of Pairwise Distances as the Cost Function

Our algorithms can also be applied to the situation where the sum of all pairwise distances is used as the cost function, that is,

$$f(C) = \binom{|C|}{2} \cdot \max_{o, o' \in C} dist(o, o') \qquad (4)$$

TABLE I. The statistics of the real datasets

Dataset	UK	FSQ
Number of objects	179,491	512,590
Avg. # keywords/object	6.50	4.69
Number of unique keywords	117,305	166,909

Only a small change in both *PolyLune* and *ScaleLune* is needed. For both *PolyLune* and *ScaleLune*, we can just use the cost function in Equation 4 in Lines 8 and 9 in Algorithm 1 and Lines 22 and 23 in Algorithm 2. In addition, for *ScaleLune*, we need to change all the parts related to our pruning rules such as replacing $k_{LB}-1$ by $\binom{k_{LB}}{2}$ and replacing $\lceil \frac{|T|}{w} - 1 \rceil$ by $\binom{\lceil \frac{|T|}{w} \rceil}{2}$.

It is easy to see that these changes in our algorithms do not impair their correctness and complexity, but their approximation bounds are adjusted as follows.

Theorem 5: The *PolyLune* (or *ScaleLune*) algorithm returns an $O(\log^2 |T|)$-approximate answer for SK-Cover with respect to the distance cost function in Equation 4.

Proof: Let C^* be the optimal group and $\hat{C}$ be the group returned by *PolyLune* (or *ScaleLune*). Similar to the proof of Lemma 3, we have:

$$\begin{aligned} f(\hat{C}) &\leq f(X^*) \leq \sqrt{3} \cdot r^* \cdot \binom{|X^*|}{2} \\ &\leq \sqrt{3} \cdot (\ln |T| + 1)^2 \cdot \frac{|C^*|}{|C^*|-1} \cdot r^* \cdot \binom{|C^*|}{2} \\ &= O(\log^2 |T|) \cdot f(C^*) \end{aligned}$$

■

Similarly, we can obtain that the straightforward adaptations from solutions for mCK or Set-Cover that are explained in Section III-C now have an $O(|T|^2)$-approximation bound for this distance cost function. The gap between the bounds of the adapted algorithms and ours in this case is even bigger than that when Equation 1 is used, since $\frac{|T|}{\log |T|} \leq (\frac{|T|}{\log |T|})^2$.

V. Experiments

In this section, we report experimental results on the effectiveness and efficiency of our methods, and also compare with the baseline methods.

A. Setup

1) Datasets: Our experiments were conducted on two real datasets, whose statistics are shown in Table I. Dataset UK[1] is the set of POIs (e.g., hospital, supermarket, park, etc.) of the United Kingdom, where each POI is augmented with a simple textual description. It consists of about 0.2 million POIs and one million words. Dataset FSQ is a set of 0.5 million *venues* in New York crawled from *Foursquare*[2]. Each venue is associated with a GPS location and contains some textual attributes such as its name and category. For both real datasets, we normalized all the spatial coordinates to range $[0, 1]$.

2) Queries: To construct queries in our experiments, we consider the following factors. First, we vary the number of query keywords (i.e., $|T|$) from 2 to 20 (10 by default). Also, we consider the *keyword frequency rate*, that is, the percentage of objects containing a keyword in the dataset (i.e., $\frac{|O_t|}{|O|}$, where O_t is the set of objects containing keyword t). The maximum keyword frequency rates in both datasets are around 0.1. Therefore, we vary the minimum frequency rate for query keywords from 0.0025 to 0.04. Among all the keywords passing the frequency rate threshold, we randomly choose $|T|$ keywords evenly to form a query.

TABLE II. The parameters used to generate queries in the experiments

Parameter	Values **(default value)**
$\|T\|$	2, 4, 6, 8, **(10)**, 12, 14, 16, 18, 20
$\min_{t \in T} \frac{\|O_t\|}{\|O\|}$	0.0025, 0.05, **(0.01)**, 0.02, 0.04

Table II shows the parameter settings used in generating the queries in our experiments. For each configuration, we randomly produced 50 queries and report the average results.

3) Algorithms: We implemented the following algorithms and compared their performance in terms of the approximation quality and the efficiency.

- *GKG* [7] - This is a greedy 2-approximation algorithm for mCK.
- *SKECa*$^+$ [7] - This is a state-of-the-art approximation algorithm for mCK, which turns out to be slower than *GKG* but gives a more accurate answer. This algorithm basically finds the smallest enclosing circle covering all the query keywords. We use the same default parameter configuration as [7].
- *GreedyMinSK* - This algorithm adapts *GreedySetCover* to solve SK-Cover as explained in Section III-C. It first selects an object containing the maximum number of query keywords as the starting point. Iteratively it chooses the next object in a greedy manner considering both the covering power and the incurred distance. Since there can be multiple objects equally containing the maximum number of query keywords, we do the same process starting with each of them and choose the resulting set with the minimum cost.
- *PolyLune* - This is our polynomial time approximation algorithm presented in Section IV-A without using any pruning techniques and the polar-tree.
- *2DLune* - This algorithm employs all our pruning techniques presented in Section IV-B2 but not the polar-tree. It performs a 2D range search on the virtual bR*-tree to test and process each lune.
- *ScaleLune* - This is our ultimate approximation algorithm employing all the techniques including the polar-tree.

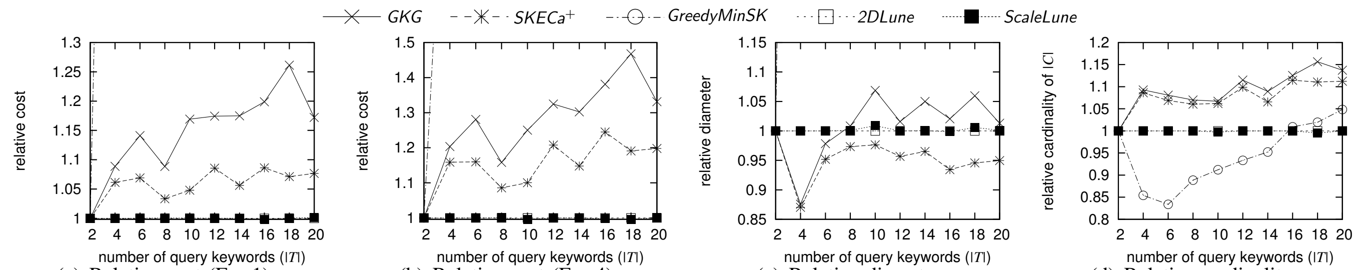
Indeed, the algorithms for mCK do not return a compact set of objects because they do not consider the cardinality of the resulting set. For instance, *SKECa*$^+$ sometimes returns a huge set containing even a lot more than $|T|$ objects. Therefore, for a more reasonable comparison, we gave all alternative algorithms an additional advantage by running *GreedySetCover* on the resulting set, and thereby the answer set can contain at most $|T|$ objects.

4) Experimental Environment: We implemented all algorithms in Java, and conducted all experiments on a PC running Linux (CentOS 6.7) equipped with Intel Core i7 CPU 3.6GHz and 16GB memory.

B. Accuracy

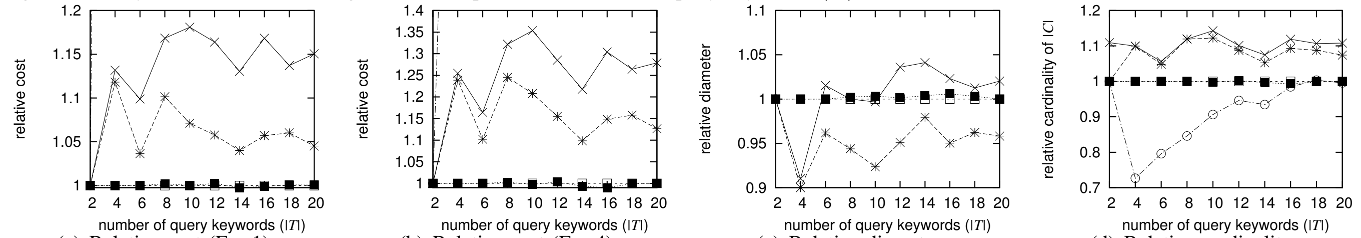
To evaluate the accuracy of each algorithm, we compare the distance costs of the answers returned from all the algo-

[1]http://www.pocketgpsworld.com
[2]https://foursquare.com



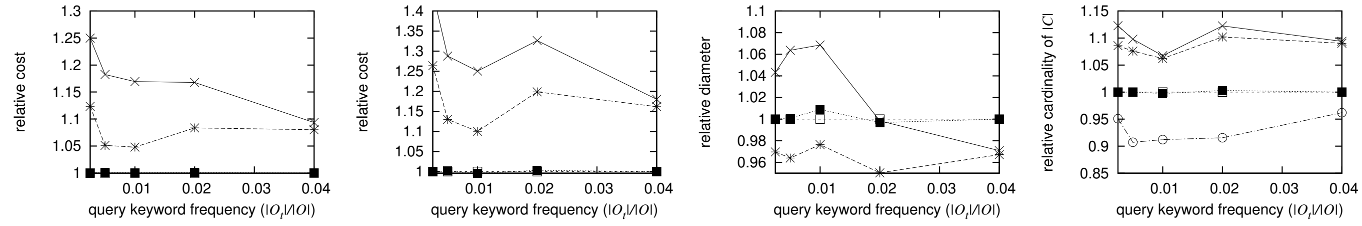
(a) Relative cost (Eq. 1) (b) Relative cost (Eq. 4) (c) Relative diameter (d) Relative cardinality

Fig. 7. Quality of the answer set using UK with respect to the number of query keywords ($|T|$)



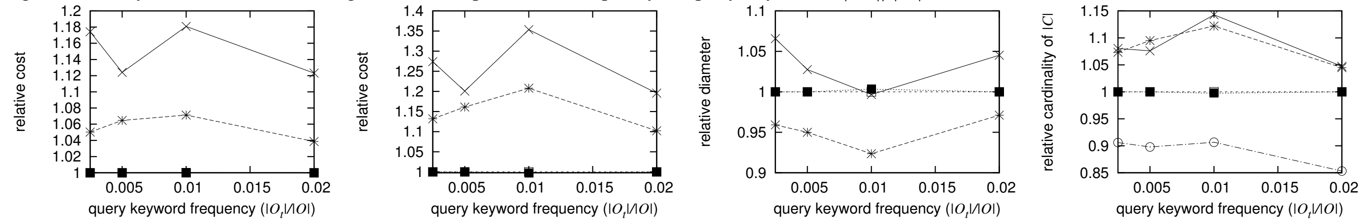
(a) Relative cost (Eq. 1) (b) Relative cost (Eq. 4) (c) Relative diameter (d) Relative cardinality

Fig. 8. Quality of the answer set using FSQ with respect to the number of query keywords ($|T|$)



(a) Relative cost (Eq. 1) (b) Relative cost (Eq. 4) (c) Relative diameter (d) Relative cardinality

Fig. 9. Quality of the answer set using UK with respect to the frequency of query keywords ($|O_t|/|O|$)



(a) Relative cost (Eq. 1) (b) Relative cost (Eq. 4) (c) Relative diameter (d) Relative cardinality

Fig. 10. Quality of the answer set using FSQ with respect to the frequency of query keywords ($|O_t|/|O|$)

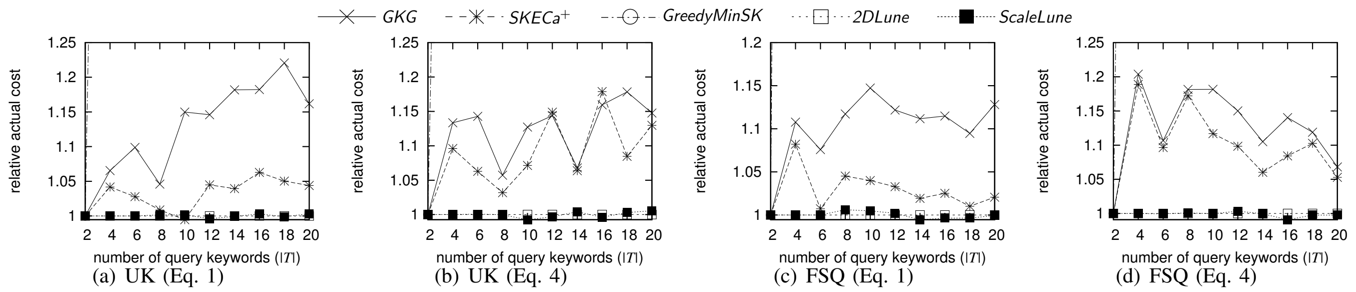
rithms. Since SK-COVER is even harder than SET-COVER, it is impractical to obtain the exact answer for SK-COVER. Therefore, we use the results from *2DLune* as the benchmark and calculate the relative cost, including relative diameter and relative cardinality of the answer sets, for the results returned from the other algorithms.

For the sake of simplicity, we only report the results from our algorithms optimizing distance cost using Equation 1, but also report the distance cost of those results against Equation 4. When evaluating using cost function Equation 4, it is a clear handicap to our methods. Interestingly, our algorithms still surpass the other competitors with a clear margin.

1) Effect of the Number of Query Keywords: Figures 7 and 8 show the relative costs with respect to cost function Equations 1 and 4, respectively, as well as the relative diameter and the relative cardinality with respect to the number of query keywords (i.e., $|T|$). On both datasets, we observe that *2DLune* and *ScaleLune* always achieve the best accuracy and their results are almost the same. Our *GreedyMinSK* algorithm returns the most inaccurate results whose cost is at least several times larger than those from *2DLune* and *ScaleLune*. *GreedyMinSK* tends to focus on minimizing the cardinality rather than the diameter as shown in Figures 7(d) and 8(d). *SKECa*$^+$ always returns the groups with a smallest diameter, which leads to a better accuracy than *GKG*. However, in terms of the cost in both Equations 1 and 4, *SKECa*$^+$ is always less accurate than *ScaleLune* because the cardinality of the resulting group (i.e., $|C|$) is always larger than those from *ScaleLune*.

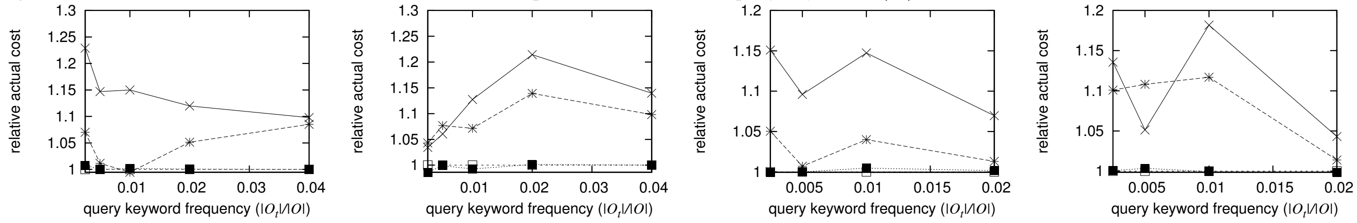
The gaps in accuracy between *ScaleLune* and the others become larger when $|T|$ increases, which is consistent with the fact that *GKG* and *SKECa*$^+$ can result in $O(|T|)$-approximation in the worst case. Also, the gaps become even lager when Equation 4 is used, as analyzed in Section IV-D.

2) Effect of the Frequency of Query Keywords: Figures 9 and 10 show the experimental results with respect to the minimum frequency rate of query keywords (i.e., $\min_{t \in T} \frac{|O_t|}{|O|}$). For the FSQ dataset, we ignored the frequency rate of 0.4



(a) UK (Eq. 1) (b) UK (Eq. 4) (c) FSQ (Eq. 1) (d) FSQ (Eq. 4)

Fig. 11. The actual distance cost of the answer set with respect to the number of query keywords ($|T|$)



(a) UK (Eq. 1) (b) UK (Eq. 4) (c) FSQ (Eq. 1) (d) FSQ (Eq. 4)

Fig. 12. The actual distance cost of the answer set with respect to the frequency of query keywords ($|O_t|/|O|$)

since only one query was possible with the frequency rate of 0.4. Similar to Figures 7 and 8, the overall accuracy of *2DLune* and *ScaleLune* is the best among all competitors. It again confirms that *GreedyMinSK* pursues the minimum cardinality while *GKG* and *SKECa*$^+$ attempt to minimize the diameter. *ScaleLune* achieves a good placement between these two objectives. One interesting observation is that *GKG* gets better with higher frequency rates. This is probably because *GKG* has to check many candidate sets when the smallest frequency is large and this leads to a better accuracy.

3) Effectiveness of the Cost Functions: Even though our algorithms are the best with respect to the cost functions in SK-COVER, we still want to investigate how effective they are in reducing the actual distance costs. We report the actual distance costs of all the retrieved groups in both Equations 1 and 4 in Figures 11 and 12. For the actual cost in Equation 1, we compute the sum of the distances from one object to the others and report the worst case. For the actual cost in Equation 4, we sum up all pairwise distances in the group.

Once again, *2DLune* and *ScaleLune* achieve the minimum distance cost in almost every case and *GreedyMinSK* is dramatically worse than the others.

Interestingly, in this experiment, the gap between *SKECa*$^+$ and *GKG* using Equation 4 gets much smaller than that using Equation 1. This is because the cardinality of answers returned by *SKECa*$^+$ is almost the same as that by *GKG*. The results indicate that the cardinality is a more important factor than the diameter to determine the actual cost in Equation 4. By minimizing the cardinality as well as the diameter, our algorithms almost always return the group of the smallest distance cost in both Equation 1 and Equation 4.

C. Efficiency

In order to evaluate the efficiency of our algorithms, we compare the execution time of all the algorithms.

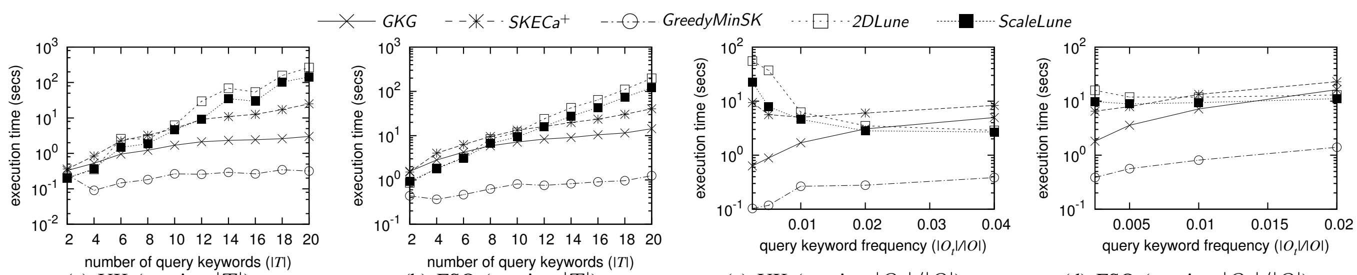
1) Overall Efficiency: The overall efficiency of our algorithms is shown in Figure 13. The algorithms based on a greedy scheme, *GKG* and *GreedyMinSK*, run faster than the other algorithms. *GreedyMinSK* is always the fastest probably because of sacrificing the accuracy a lot. *GKG* is also fast when the keyword frequency rate is low. However, its running time increases as the frequency rate increases, and becomes even slower than *2DLune* and *ScaleLune*. This is due to the fact that the more objects having the most infrequent keyword, the more candidate sets *GKG* needs to consider. *ScaleLune* is always faster than *2DLune*, and comparable with *SKECa*$^+$. This is somewhat surprising in that *SKECa*$^+$ is actually intended for mCK that turns out to be computationally easier than SK-COVER.

2) Effectiveness of the Pruning Techniques: To study the pruning power of our techniques presented in Section IV-B2, we compare the execution times of *PolyLune* and *ScaleLune*. Unfortunately, *PolyLune* is way too slow to process the real datasets. Therefore, we have to use a sample of $10,000$ objects from the UK dataset, denoted by TinyUK. Figure 14 shows the execution time of *PolyLune*, *2DLune*, and *ScaleLune* on the TinyUK dataset. Clearly, our pruning techniques substantially reduce the search space. There is a huge performance gap between *PolyLune* and *ScaleLune*.

3) Effectiveness of the Polar-tree: Figure 15 shows the speedup of *ScaleLune* over *2DLune*. Note that both algorithms consider the same number of pairs that are not pruned by our pruning techniques. The only difference is how to test whether a lune covers T and how to retrieve all the objects in the lune. We obtain speedup values of 1.9 and 1.3 on average (4.7 and 1.6 at most) on the datasets UK and FSQ, respectively. Also, the performance gain of *ScaleLune* becomes larger when the frequency of query keywords gets smaller or more query keywords need to be covered. This is because there are less groups covering T in those cases, and the early test on covering T in *ScaleLune* can successfully avoid the overhead of retrieving all the objects in checking the corresponding lune. This is also consistent with the output-sensitive characteristic of *ScaleLune* as analyzed in Section IV-C.

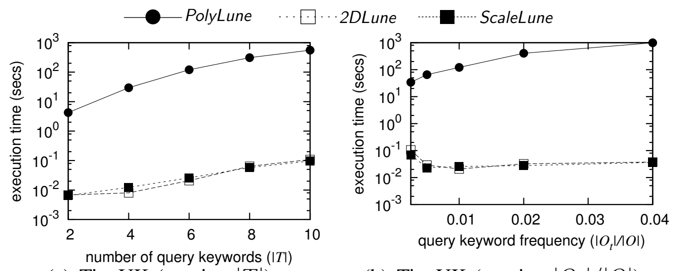
VI. CONCLUSIONS

In this paper, motivated by the observation that mCK may not capture the information need in some application scenarios, we propose the SK-COVER problem. Theoretically, we prove the NP-hardness and the approximability of SK-COVER, and



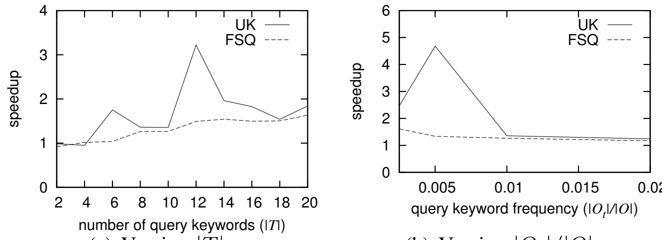
(a) UK (varying $|T|$) (b) FSQ (varying $|T|$) (c) UK (varying $|O_t|/|O|$) (d) FSQ (varying $|O_t|/|O|$)

Fig. 13. Overall efficiency test with respect to the number of query keywords ($|T|$) and the frequency of query keywords ($|O_t|/|O|$)



(a) TinyUK (varying $|T|$) (b) TinyUK (varying $|O_t|/|O|$)

Fig. 14. Efficiency test on *PolyLune* and *ScaleLune*



(a) Varying $|T|$ (b) Varying $|O_t|/|O|$

Fig. 15. Speedup of *ScaleLune* using the polar-tree

develop a polynomial time $O(\log |T|)$-approximation algorithm, which is the optimal in terms of the approximability. Practically, we devise two pruning schemes to reduce the search space, and propose a polar-tree structure to improve the efficiency of processing pairs of objects. According to our extensive experimental results, our algorithms achieve the best accuracy and a competent efficiency comparable to a state-of-the-art mCK query processing algorithm. As future work, it is interesting to devise an exact and practically feasible algorithm for SK-COVER even though it could be a tough challenge. Moreover, it is desirable to consider more distance cost functions addressing different needs in applications.

ACKNOWLEDGMENT

Choi and Pei's research is supported in part by an NSERC Discovery grant, the Canada Research Chair program, and a Yahoo! Faculty Research and Engagement Program (FREP) award. Lin's research is supported in part by NSFC61232006, ARC DP140103578 and ARC DP150102728. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa, "Keyword search in spatial databases: Towards searching by document," in *ICDE*, 2009.

[2] D. Zhang, B. C. Ooi, and A. K. H. Tung, "Locating mapped resources in web 2.0," in *ICDE*, 2010.

[3] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi, "Collective spatial keyword querying," in *SIGMOD*, 2011.

[4] L. Zhang, X. Sun, and H. Zhuge, "Density based collective spatial keyword query," in *SKG*, 2012.

[5] C. Long, R. C. Wong, K. Wang, and A. W. Fu, "Collective spatial keyword queries: a distance owner-driven approach," in *SIGMOD*, 2013.

[6] K. Deng, X. Li, J. Lu, and X. Zhou, "Best keyword cover search," *IEEE TKDE*, vol. 27, no. 1, 2015.

[7] T. Guo, X. Cao, and G. Cong, "Efficient algorithms for answering the m-closest keywords query," in *SIGMOD*, 2015.

[8] R. Fleischer and X. Xu, "Computing minimum diameter color-spanning sets," in *FAW*, 2010.

[9] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, vol. 4, no. 3, 1979.

[10] C. Lund and M. Yannakakis, "On the hardness of approximating minimization problems," *Journal of the ACM*, vol. 41, no. 5, 1994.

[11] I. D. Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," in *ICDE*, 2008.

[12] A. Cary, O. Wolfson, and N. Rishe, "Efficient and scalable method for processing top-k spatial boolean queries," in *SSDBM*, 2010.

[13] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen, "Joint top-k spatial keyword query processing," *IEEE TKDE*, vol. 24, no. 10, 2012.

[14] Y. Tao and C. Sheng, "Fast nearest neighbor search with keywords," *IEEE TKDE*, vol. 26, no. 4, 2014.

[15] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *PVLDB*, vol. 2, no. 1, 2009.

[16] Z. Li, K. C. K. Lee, B. Zheng, W. Lee, D. L. Lee, and X. Wang, "Ir-tree: An efficient index for geographic document search," *IEEE TKDE*, vol. 23, no. 4, 2011.

[17] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg, "Efficient processing of top-k spatial keyword queries," in *SSTD*, 2011.

[18] M. Abellanas, F. Hurtado, C. Icking, R. Klein, E. Langetepe, L. Ma, B. Palop, and V. Sacristán, "The farthest color voronoi diagram and related problems," in *EWCG*, 2001.

[19] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. Teng, "On trip planning queries in spatial databases," in *SSTD*, 2005.

[20] X. Ma, S. Shekhar, H. Xiong, and P. Zhang, "Exploiting a page-level upper bound for multi-type nearest neighbor queries," in *GIS*, 2006.

[21] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi, "The optimal sequenced route query," *VLDB J.*, vol. 17, no. 4, 2008.

[22] H. Chen, W. Ku, M. Sun, and R. Zimmermann, "The multi-rule partial sequenced route query," in *GIS*, 2008.

[23] J. Li, Y. D. Yang, and N. Mamoulis, "Optimal route queries with arbitrary order constraints," *IEEE TKDE*, vol. 25, no. 5, 2013.

[24] G. Cormode, H. J. Karloff, and A. Wirth, "Set cover algorithms for very large datasets," in *CIKM*, 2010.

[25] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi, "The priority r-tree: A practically efficient and worst-case optimal r-tree," *ACM TALG*, vol. 4, no. 1, 2008.

[26] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry 3rd revised ed.* Springer-Verlag, 2008.