Collective Spatial Keyword Querying

Xin Cao[1] Gao Cong[1] Christian S. Jensen[2] Beng Chin Ooi[3]

[1]School of Computer Engineering, Nanyang Technological University, Singapore
xcao1@e.ntu.edu.sg, gaocong@ntu.edu.sg

[2]Department of Computer Science, Aarhus University, Denmark
csj@cs.au.dk

[3]School of Computing, National University of Singapore, Singapore
ooibc@comp.nus.edu.sg

ABSTRACT

With the proliferation of geo-positioning and geo-tagging, spatial web objects that possess both a geographical location and a textual description are gaining in prevalence, and spatial keyword queries that exploit both location and textual description are gaining in prominence. However, the queries studied so far generally focus on finding individual objects that each satisfy a query rather than finding groups of objects where the objects in a group collectively satisfy a query.

We define the problem of retrieving a group of spatial web objects such that the group's keywords cover the query's keywords and such that objects are nearest to the query location and have the lowest inter-object distances. Specifically, we study two variants of this problem, both of which are NP-complete. We devise exact solutions as well as approximate solutions with provable approximation bounds to the problems. We present empirical studies that offer insight into the efficiency and accuracy of the solutions.

1. INTRODUCTION

With the proliferation of geo-positioning, e.g., by means of GPS or systems that exploit the wireless communication infrastructure, accurate user location is increasingly available. Similarly, increasing numbers of objects are available on the web that have an associated geographical location and textual description. Such *spatial web objects* include stores, tourist attractions, restaurants, hotels, and businesses.

This development gives prominence to *spatial keyword queries* [5, 6, 8, 10]. A typical such query takes a location and a set of keywords as arguments and returns the single spatial web object that best matches these arguments.

We observe that user needs may exist that are not easily satisfied by a single object, but where groups of objects may combine to meet the user needs. Put differently, the objects in a group collectively meet the user needs. For example, a tourist may have particular shopping, dining, and accommodation needs that may best be met by several spatial web objects. As another example, a user may wish to set up a project consortium of partners within a certain spatial proximity that combine to offer the capabilities required for the successful execution of the project.

To address the need for such collective answers to spatial keyword queries, we assume a database of spatial web objects and then consider the problem of how to retrieve a group of spatial objects that collectively meet the user's needs given as location and a set of keywords: 1) the textual description of the group of objects must cover the query keywords, 2) the objects are close to the query point, and 3) the objects in the group are close to each other.

Specifically, given a set of spatial web objects D, and a query $q = (\lambda, \psi)$, where λ is a location and ψ is a set of keywords, we consider two instantiations of the *spatial group keyword query*.

1. We aim to find a group of objects χ that cover the keywords in q such that the sum of their spatial distances to the query is minimized.

2. We aim to find a group of objects χ that cover the keywords in q such that the sum of the maximum distance between an object in χ and query q and the maximum distance between two objects in χ is minimized.

It turns out that the subproblems corresponding to these two instantiations are both NP-complete. The first subproblem can be reduced from the weighted set cover problem. We propose a greedy algorithm that provides an approximate solution to the problem. This algorithm utilizes a spatial-keyword index such as the IR-tree [8] to prune the search space. The algorithm has a provable approximation bound. Based on the assumption that in some applications, the number of keywords in a query q may not be large, we also propose an exact algorithm that exploits dynamic programming and a spatial-keyword index. The exact algorithm avoids enumerating the combinations of data objects in the database. Rather, it enumerates the query keywords and exploits a series of pruning strategies to reduce the search space.

For the second subproblem, we develop two approximation algorithms based on a spatial-keyword index with provable approximation bounds. The first approximation algorithm has a 3-approximation ratio, while the second algorithm has a 2-approximation ratio. We also develop an exact algorithm that exploits a spatial-keyword index and the geometric property and branch-and-bound search to prune the search space.

In summary, our contribution is threefold. First, we propose a new type of queries, called *spatial group keyword queries*, that find groups of objects that collectively satisfy a query. We consider two instantiations of the problem and show that both are NP-complete.

Second, we propose algorithms that offer approximate solutions to the two subproblems with provable approximation bounds. We

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

also present exact algorithms for the two subproblems. All algorithms exploit a spatial keyword index to prune the search space.

Third, we study the properties of the paper's proposals empirically based on prototype implementations of the proposals. The results demonstrate that the proposals offer scalability and are capable of excellent performance.

The rest of the paper is organized as follows. Section 2 formally defines the problem and establishes the computational complexities of the problem. Section 3 presents an approximate algorithm and an exact algorithm for the first subproblem. Section 4 presents two approximate algorithms and an exact algorithm for the second subproblem. We report on the empirical studies in Section 5. Finally, we cover related work in Section 6 and offer conclusions and research directions in Section 7.

2. PROBLEM STATEMENT

Let S be a set of keywords. The keywords may capture user preferences or required project partner capabilities, depending on the application. Let D be a database consisting of m spatial web objects. Each object o in D is associated with a location $o.\lambda$ and a set of keywords $o.\psi$, $o.\psi \subset S$, that describe the object (e.g., the menu of restaurant or the skills of a possible project partner).

Consider a spatial group keyword query $q = \langle q.\lambda, q.\psi \rangle$, where $q.\lambda$ is a location and $q.\psi$ represents a set of keywords. The *spatial group keyword query* finds a group of objects χ, $\chi \subseteq D$, such that $\cup_{r\in\chi} r.\psi \supseteq q.\psi$ and such that the cost $\mathsf{Cost}(\chi)$ is minimized.

We proceed to present cost functions. Given a set of objects χ, the cost function has two weighted components:

$$\mathsf{Cost}(q,\chi) = \alpha C_1(q,\chi) + (1-\alpha)C_2(\chi),$$

where $C_1(\cdot)$ is dependent on the distance of the objects in χ to the query object and $C_2(\cdot)$ characterizes the inter-object distances among the objects in χ.

This type of cost function is capable of expressing that result object should be near the query location ($C_1(\cdot)$), that the result objects should be near to each other ($C_2(\cdot)$, and that these two aspects are given different weights (α). We consider two instantiations of the cost function $\mathsf{Cost}(q,\chi)$ that we believe match the intended applications well.

TYPE1 cost function:

$$\mathsf{Cost}(q,\chi) = \sum_{r\in\chi}(\mathsf{Dist}(r,q)) \quad (1)$$

The cost function is the sum of the distance between each object in χ and the query location. This may fit with applications where the objects need to meet at the query location, such as incident handling or the finding of project partners.

TYPE2 cost function:

$$\mathsf{Cost}(q,\chi) = \alpha \max_{r\in\chi}(\mathsf{Dist}(r,q)) + (1-\alpha)\max_{r_1,r_2\in\chi}(\mathsf{Dist}(r_1,r_2)) \quad (2)$$

The first part of this cost function is the maximal distance between any object in χ and the query location q, and the second part is the maximum distance between two objects in χ (this can be understood as the diameter of the result). When there are multiple optimal groups of objects, we choose one group randomly. This second cost function may fit with applications such as tourists planning visits to several points of interest.

For ease of presentation, we disregard parameter α in the rest of this paper. But the proposed algorithms remain applicable when α is enabled.

Lemma 2.1: *The decision version of spatial group keyword query problem using either a* TYPE1 *or a* TYPE2 *cost function is NP-complete.*

Proof: *We first consider the* TYPE1 *cost function. We prove the lemma by reduction from the weighted set cover problem. An instance of the weighted cover problem consists of a universe U = { 1, 2, ..., n} of n elements and a family of sets $\mathcal{S} = \{S_1, S_2, ..., S_m\}$, where $S_i \subseteq U$ and each S_i is associated with a positive cost C_{S_i}. The decision problem is to decide if we can find a subset $\mathcal{F}$ of $\mathcal{S}$ such that $\cup_{S_i\in\mathcal{F}} S_i$ = U and such that its cost $\sum_{S_i\in\mathcal{F}}(C_{S_i})$ is minimized.*

To reduce this problem to the TYPE1 *spatial group keyword query problem, we observe thateach element in U corresponds to a keyword in $q.\phi$, that each S_i corresponds to a spatial object containing a set of keywords, and that the weight of S_i is $dist(q, S_i)$. It is easy to show that there exists a solution to the weighted set cover problem if and only if there exits a solution to query q. This completes the proof.*

Considering next the TYPE2 *cost function, we prove the lemma by reduction from the 3-SAT problem. An instance of the 3-SAT problem consists of $\Phi = C_1 \wedge C_2, ..., \wedge C_l$, where each clause C_j = $x_j \vee y_j \vee z_j$, and $\{x_j, y_j, z_j\} \in \{e_1, \bar{e_1}, e_2, \bar{e_2}, ..., e_n, \bar{e_n}\}$. The decision problem is to determine whether we can assign a truth value to each of the literals (e_1 through e_n) such that Φ is* true.

Next, we reduce this problem to an instance of the TYPE2 *spatial group keyword query problem. The reduction is inspired by the proof for the hardness of the Multiple-Choice cover problem [2] (that is different from our problem). Consider a circle with diameter d and with the query point q as its center, and let each variable e_i correspond to a point in the circle, while its negation $\bar{e_i}$ corresponds to the diametrically opposite on the circle. The distance between e_i and $\bar{e_i}$ is d. We choose d such that $d > d_1$ is sufficiently close to d_1; thus, the distance between any two points corresponding to different variables are smaller than d_1.*

Each set S_i ($i \in [1, n]$) contains a pair of points e_i and $\bar{e_i}$, and the two points contain a distinct keyword in $q.\psi$. Each set S_j (j $\in [n + 1, n + l]$) contains each triple of points corresponding to a cause C_{j-n}, and they contain a distinct keyword in $q.\psi$. Thus, to cover all keywords in $q.\psi$, a query result of q must contain one point from each S_i (i.e., e_i and $\bar{e_i}$), and it must contain at least one point from each S_j (corresponding to clause C_{j-n}).

Given this mapping, we can see that if there exits a truth assignment for Φ, all the keywords in q are covered, and $max_{r\in\chi}\mathsf{dist}(q,r)$ + $max_{i,j\in\chi}\mathsf{dist}(i,j)$ = $\frac{d}{2} + max\mathsf{dist}(i,j), i,j \in \chi$ is minimized, i.e., a feasible solution χ with at most $\frac{d}{2} + d_1$ exists. On the other hand, if there exists a subset of points on the circle whose diameter is at most d_1, there exists a truth assignment for the instance Φ. This completes the proof. □

In the problem definition, we assume that each object either has or does not have a given keyword. In some applications, e.g., where keywords represent skills, degrees can be associated with the keywords, and queries then require certain minimum degrees for their keywords. To accommodate this setting, we assume that an object has a keyword only if its keyword degree is equal to or higher than the degree required by the query at hand.

3. PROCESSING OF TYPE1 SPATIAL GROUP KEYWORD QUERIES

3.1 Preliminaries: The IR-Tree

We briefly review the IR-tree [8], which we use as an index structure in the algorithms to be presented. We note that other spatial-keyword indexes (e.g., [10]) may be used in its place.

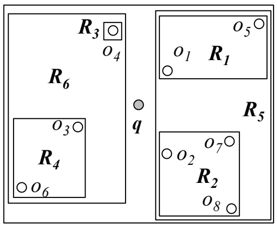
The IR-tree [8] is essentially an R-tree [12] extended with inverted files [16]. Each leaf node in the IR-tree contains entries of the form $(o, o.\lambda, o.di)$, where o refers to an object in dataset D, $o.\lambda$ is the bounding rectangle of o, and $o.di$ is an identifier of the description of o. Each leaf node also contains a pointer to an inverted file with the keywords of the objects stored in the node.

An inverted file index has two main components.

- A vocabulary of all distinct words appearing in the description of an object.
- A posting list for each word t that is a sequence of identifiers of the objects whose descriptions contain t.

Each non-leaf node R in the IR-tree contains a number of entries of the form $(cp, rect, cp.di)$, where cp is the address of a child node of R, $rect$ is the minimum bounding rectangle (MBR) of all rectangles in entries of the child node, and $cp.di$ is an identifier of a pseudo text description that is the union of all text descriptions in the entries of the child node.

As an example, Figure 1(a) contains eight spatial objects $o_1, o_2, \ldots, o_8$, and Figure 1(b) shows the words appearing in the description of each object. Figure 2 illustrates the corresponding IR-tree, and Table 1 shows the content of the inverted files associated with the nodes.



(a) object locations

object	words
o_1	t_1, t_2
o_2	t_2, t_3
o_3	t_1, t_3
o_4	t_1, t_5
o_5	t_2, t_4
o_6	t_4, t_6
o_7	t_4, t_5
o_8	t_1, t_5

(b) object descriptions

Figure 1: A Dataset of Spatial Keyword Objects

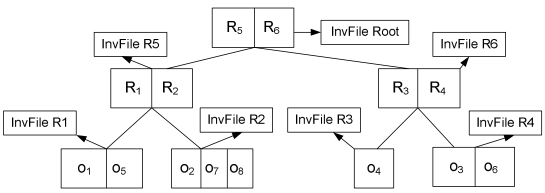


Figure 2: Example IR-Tree

Table 1: Content of Inverted Files of the IR-Tree

$Root$	R_5	R_6	R_1	R_2	R_3	R_4
t_1: R_5, R_6	t_1: R_1, R_2	t_1: R_3, R_4	t_1: o_1	t_1: o_8	t_1: o_4	t_1: o_3
t_2: R_5	t_2: R_1, R_2	t_3: R_4	t_2: o_1, o_5	t_2: o_2	t_5: o_4	t_3: o_3
t_3: R_5, R_6	t_3: R_2	t_4: R_4	t_4: o_5	t_3: o_2		t_4: o_6
t_4: R_5, R_6	t_4: R_1, R_2	t_5: R_3		t_4: o_7		t_6: o_6
t_5: R_5, R_6	t_5: R_2	t_6: R_4		t_5: o_7, o_8		
t_6: R_6						

3.2 Approximation Algorithm

We show that the first subproblem is NP-complete by a reduction from the Weighted Set Cover (WSC) problem in Lemma 2.1. The reduction in the proof is approximation preserving. Thus, the approximation properties of the WSC problem carry over to our problem.

For the WSC problem, it is known (see [7]) that a greedy algorithm is an H_k-approximation algorithm for the weighted k-set cover, where $H_k = \sum_{i=1}^{k} \frac{1}{i}$ is the k-th harmonic number. In our problem, k is the number of query keywords. Thus, we can adapt the greedy algorithm to process the TYPE1 spatial group keyword query.

A straightforward method of adapting the greedy algorithm is to decompose the given user query q dynamically into a sequence of partial queries, each containing a different set of keywords depending on the preceding partial queries, and then to evaluate these partial queries. Specifically, we start with the user query q, which can be regarded as the first partial query, and we find the object with the lowest cost that covers part or all of the keywords in q. The object is added to the result set. The uncovered keywords in q form a new partial query with the same spatial location as q. We then find an object with the lowest cost that covers part or all of the keywords in the new partial query. This process continues until all keywords are covered or some keyword cannot be covered by any object. This method needs to scan the dataset multiple times, once for each partial query.

To avoid multiple scans, we propose a greedy algorithm on top of the IR-tree. We proceed to focus on two aspects of the idea that are important to the performance: (1) how to find the object with the lowest cost for each partial query using the IR-tree, and (2) whether we can reuse the computation for the preceding partial query when computing the next partial query.

Given a partial query q_s, we adopt the best-first strategy to traverse the IR-tree. We use a min-priority queue to maintain the intermediate results. The key of the queue is the cost of each element. The cost of an object o is computed by $\frac{\mathsf{Dist}(q,o)}{|o.\psi \cap q_s.\psi|}$; the cost of a node entry e is computed by $\frac{\mathsf{minDist}(q,e)}{|e.\psi \cap q_s.\psi|}$, where $\mathsf{minDist}$ (q, e) represents the minimum distance between q and e.

Lemma 3.1: *Given a partial query q_s and an IR-tree, the cost of a node is a lower bound of the cost of any of its child nodes.*

Proof: *Given a node e and any of its child nodes e', we have* $\mathsf{minDist}$ $(q, e) \leq \mathsf{minDist}$ (q, e'), *and* $|e.\psi \cap q_s.\psi| \geq |e'.\psi \cap q_s.\psi|$. □

Lemma 3.1 says that the cost of a node is a lower bound of the costs of all objects in the subtree rooted at the node. Thus, if the cost exceeds that of some object that has been visited, we can disregard all objects in the subtree for q_s. This guarantees the correctness of the best-first strategy for finding an object with the lowest cost for a partial query q_s.

We next discuss whether we can reuse the computation for preceding partial queries. An obvious method is to process each partial query from scratch. However, this incurs repeated computation when a node or an object is visited for multiple times. To avoid this, we divide the entries (corresponding to leaf and non-leaf nodes) in the priority queue into two parts: (1) the entries that have already been visited when processing previous partial queries, and (2) the entries that have not yet been visited.

Lemma 3.2: *The elements in the priority queue that have been visited when processing previous partial queries can be disregarded when processing a new partial query.*

Proof: *The keyword set of a previous partial query is a superset of the keyword set of a new partial query. For a visited node, all its entries containing keywords of the new partial query have been enqueued into the priority queue; thus, we can disregard the elements that have been visited.* □

The pseudocode is outlined in Algorithm 1. The algorithm uses a min-priority queue for the best-first search with the cost as the key. Variable $mSet$ keeps the keyword set of the current partial query, and $pSet$ keeps the keyword set of the preceding partial query. For each partial query, we use the best-first search to find an object that

Algorithm 1: Type1Greedy(q, $irTree$)

```
1  U ← new min-priority queue;
2  U.Enqueue(irTree.root, 0);
3  V ← ∅; Cost ← 0;
4  mSet ← q.ψ; pSet ← q.ψ;
5  while mSet ≠ ∅ do
6      while U is not empty do
7          e ← U.Dequeue();
8          Cost ← Cost + e.Key;
9          if e is an object then
10             V ← V ∪ e;
11             pSet ← mSet ;
12             mSet ← mSet \ e.ψ;
13             for each entry e′ in U do
14                 if e′.λ∩ e.λ ≠ ∅ then
15                     e′.key = (e′.key∗|e′.λ∩mSet|) / |e′.λ∩pSet| ;
16                 else remove e from U;
17             reorganize priority queue U using new key values;
18             break;
19         else
20             read the posting lists of e for keywords in mSet;
21             for each entry e′ in node e do
22                 if mSet ∩ e′.ψ ≠ ∅ then
23                     if e is a non-leaf node then
24                         U.Enqueue(e′, minDist(q,e′) / |mSet∩e′.ψ| );
25                     else
26                         U.Enqueue(e′, Dist(q,o) / |mSet∩o.ψ|);
27 return Cost and V ;                                  // results
```

overlaps with the query keyword $mSet$ and has the lowest cost. The algorithm computes the cost for a non-object node (line 23) and the cost for an object (line 25).

Whenever the algorithm pops an object from U, it is guaranteed that the text description of the object overlaps with $mSet$ (the keyword set of the current partial query), and that the object has the lowest cost. Thus, it becomes part of the result. The algorithm proceeds with the next partial query by changing the keyword component (line 12). Based on Lemma 3.2, we do not need to scan all objects to process the new partial query. Rather, we only have to update the unvisited elements in the priority queue with the new cost based on the new partial query (lines 13–16). We then use the best-first search to process the new partial query.

3.3 Exact Algorithm

The number of keywords of a query may be small in some applications. This motivates us to develop an exact algorithm for processing the TYPE1 spatial keyword group query.

We present a dynamic programming algorithm that does not use an index in Section 3.3.1, and we present a version of the algorithm that uses an index to prune the search space in Section 3.3.2.

3.3.1 Exact Algorithm Without an Index

An obvious exact algorithm enumerates every subset of spatial objects whose text descriptions overlap with the query keyword set in D.For each such subset, the algorithm then checks whether the subset covers all query keywords and computes its cost. This yields an exponential running time in terms of the number of objects, which is very expensive if D is large.

A better method is to perform the exhaustive search on a smaller set of objects. We proceed to introduce a lemma that lays the foundation for the algorithm to be proposed.

Lemma 3.3: *Consider a query q and two objects o_i and o_j, each of which contains a subset of the query keywords. Let $ws_i = q.\psi \cap o_i.\psi$ and $ws_j = q.\psi \cap o_j.\psi$. If $\mathsf{Dist}(q, o_i) < \mathsf{Dist}(q, o_j)$, $\{o_i\}$ is a better group than $\{o_j\}$ for any keyword subset of $ws_i \cap ws_j$.*

Proof: *Obvious since o_i always incurs lower cost than does o_j for any keyword subset $ws_i \cap ws_j$.* □

According to the lemma, given a subset of query keywords ws, among the objects covering ws, the one that is the closest to the query contributes the lowest cost to ws.

Example 3.1: Consider a query q with keywords $q.\psi = \{t_1, t_2, t_3\}$ and the four objects in Table 2. We know that $\mathsf{Dist}(q, o_1) < \mathsf{Dist}(q, o_2)$ and $o_1 \cap o_2 = \{t_2\}$. According to lemma 3.3, $\{o_1\}$ is a better result set for the query with keyword set $\{t_2\}$. □

	o_1	o_2	o_3	o_4
Distance to the query	1	2	2.5	4
Keywords	t_1,t_2	t_2,t_3	t_1,t_3	t_1

Table 2: Example data set

Since the set of query keywords is small, the number of its subsets is not large although it is exponential to the number of query keywords. For each subset of query keywords, we find the object that covers the subset of query keywords and has the lowest costs according to Lemma 3.3. Then an exhaustive search on these objects can find the best group. However, this method is time-consuming since it runs exponentially in terms of the number of objects, which can be exponential in the number of query keywords.

Instead, we develop a dynamic programming algorithm with exponential running time in terms of the number of query keywords. The idea of the algorithm is summarized as follows: Given a query q with n ($= |q.\psi|$) keywords, we process the subsets of $q.\psi$ in breadth-first order, i.e., we process subsets in ascending order of their length. For each subset X of $q.\psi$, we find the best set of covering objects, i.e., a set of objects that cover X and have the lowest cost, by utilizing the best covering sets of the subsets of X.

Existing WSC algorithms are mostly approximation algorithms. Several recent proposals [4, 9] have good (e.g., constant) approximation guarantees with moderately exponential running time. Björklund et al. [3] propose an exact algorithm for the unweighted set cover problem using the inclusion-exclusion principle, which is not directly applicable to the WSC problem.

Formally, let F be the set of all subsets of $q.\psi$. For each subset $X \subseteq q.\psi$, we denote the set of objects that cover X and has the lowest cost by $\mathsf{Group}(X)$ and the cost of covering set X by $\mathsf{Cost}(X)$.

Our dynamic programming algorithm avoids enumerating all the set partitions. Equation 3 shows the approach to computing the lowest cost for each subset X. If X is not covered by any object o, its cost is initialized to ∞; otherwise, its cost is initialized to the cost of the best covering object, as shown in Equation 3. Then the dynamic programming idea is adopted to find the lowest cost of each subset X in ascending order of the length of X. Specifically, for each X, we check each pair of component subsets (whose optimal costs are already known) to find a pair with the lowest cost. Note that the optimal set of two subsets may share some objects whose costs are computed by oCost.

$$\mathsf{Cost}(X) = \begin{cases} \min_{o \in D \wedge X \subseteq o.\psi}\{\mathsf{Dist}(q, o)\}, & \exists o (X \subseteq o.\psi) \\ \infty, & \text{otherwise} \end{cases} \quad (3)$$

$$\mathsf{Cost}(X) = \min_{S \in F \wedge S \subseteq X}\{\mathsf{Cost}(X \setminus S) + \mathsf{Cost}(S) - \mathsf{oCost}(S, X \setminus S)\}$$

$$\mathsf{oCost}(S, X \setminus S) = \sum_{o \in \mathsf{Group}(X \setminus S) \cap \mathsf{Group}(S)} \mathsf{Dist}(o, q) \quad (4)$$

Algorithm 2: Type1ExactNoIndex (q, D)

1 $n \leftarrow |q.\psi|$;
2 **for** *i from* 1 *to* $2^n - 1$ **do** $\mathsf{Cost}[i] \leftarrow \infty$, $\mathsf{Group}[i] \leftarrow \emptyset$;
3 **for** *each object* o_i *in* D **do**
4   **if** $o_i.\psi \cap q.\psi \neq \emptyset$ **then**
5     $\mathsf{Dist}[o_i] \leftarrow \mathsf{Dist}\ (o_i, q)$;
6   **for** *each subset* s_i *in* $o_i.\psi \cap q.\psi$ **do**
7       $\mathsf{i} \leftarrow \mathsf{MapToInteger}(s_i)$;
8   **if** $\mathsf{Cost}[i] > \mathsf{Dist}\ [o_i]$ **then**
9         $\mathsf{Cost}[i] \leftarrow \mathsf{Dist}\ [o_i]$;
10         $\mathsf{Group}[i] \leftarrow \{o_i\}$;
11 **for** *i from* 1 *to* $2^n - 1$ **do**
12   $minValue \leftarrow \infty$, $bestSplit \leftarrow 0$;
13   **for** *j from* 1 *to* $i/2$ **do**
14   **if** $j\ \&\ i = j$ **then**
15       $S \leftarrow \mathsf{Group}[j] \bigcap \mathsf{Group}[i-j]$;
16       $oDist \leftarrow 0$;
17   **for** *each object o in* S **do**
18         $oDist \leftarrow oDist + \mathsf{Dist}\ [o]$;
19       $cost \leftarrow \mathsf{Cost}[j] + \mathsf{Cost}[i-j] - oDist$;
20   **if** $cost < minValue$ **then**
21         $minValue \leftarrow cost$;
22         $bestSplit \leftarrow j$;
23   **if** $\mathsf{Cost}[i] > minValue$ **then**
24     $\mathsf{Cost}[i] \leftarrow minValue$;
25     $\mathsf{Group}[i] \leftarrow \mathsf{Group}[bestSplit] \cup \mathsf{Group}[i - bestSplit]$;
26 **return** $\mathsf{Cost}[2^n - 1]$ and $\mathsf{Group}[2^n - 1]$

$$\mathsf{Group}(X) = \begin{cases} \arg\min_{o \in D \wedge X \subseteq o.\psi}\{\mathsf{Dist}(q,o)\}, & \exists o (X \subseteq o.\psi) \\ \emptyset, & \text{otherwise} \end{cases}$$
$$S* = \underset{S \in F \wedge S \subseteq X}{\arg\min} \{\mathsf{Cost}(X \setminus S) + \mathsf{Cost}(S) - \mathsf{oCost}(S, X \setminus S)\}$$
$$\mathsf{Group}(X) = \mathsf{Group}(S*) \cup \mathsf{Group}(X \setminus S*) \tag{5}$$

To implement the algorithm efficiently, we encode the subsets. A query q with n keywords $q.\psi = \{t_1, t_2, \ldots, t_n\}$ has $(2^n - 1)$ non-empty subsets of keywords. We encode each subset X by an integer i of n bits, where each bit corresponds to a keyword in q. If the j^{th} keyword is contained in X, the j^{th} bit in the binary format of i is set to 1; otherwise, it is set to 0. For example, for $q.\psi$=$\{t_1, t_2, t_3\}$, we can encode subset $\{t_1\}$ by 1, $\{t_2\}$ by 2, and $\{t_1, t_2\}$ by 3.

We maintain two arrays: $\mathsf{Cost}[i]$ records the lowest cost of the subset that is encoded by integer i, and $\mathsf{Group}[i]$ records the group of objects that contribute to the lowest cost. Equations 4 and 5 can be rewritten as the following equation:

$$\begin{aligned} \mathsf{Cost}[i] &= \min_{j, i\&j=j}\{\mathsf{Cost}[j] + \mathsf{Cost}[i-j] - \mathsf{oCost}(i,j)\} \\ \mathsf{oCost}(i,j) &= \sum_{o \in \mathsf{Group}[i] \cap \mathsf{Group}[i-j]} \mathsf{Dist}(o,q) \\ opt &= \underset{j, i\&j=j}{\arg\min}\{\mathsf{Cost}[j] + \mathsf{Cost}[i-j] - \mathsf{oCost}(i,j)\} \\ \mathsf{Group}[i] &= \mathsf{Group}[opt] \cup \mathsf{Group}[i - opt] \end{aligned} \tag{6}$$

Here, $\&$ is the bit-wise AND operator, and $i\&j = j$ states that the set represented by j is a subset of the set represented by i.

The pseudocode is outlined in Algorithm 2. We progressively compute $\mathsf{Cost}[i]$ and $\mathsf{Group}[i]$ from $i = 1$ to $(2^n - 1)$. When $i = (2^n - 1)$, we get the results with the lowest cost, $\mathsf{Cost}[2^n - 1]$, and the best group, $\mathsf{Group}[2^n - 1]$. We scan the dataset D. For each single object that overlaps with query keyword set $q.\psi$, we check whether the object contributes to the lowest cost for a keyword subset (lines 3–10). If a keyword subset is not contained by any single object, its cost is initialized to ∞.

The algorithm proceeds to use Equation 6 to compute the lowest cost for each subset (lines 11–25). We check each subset, represented by j, of the current keyword subset, represented by i, to see whether the subset represented by j and the subset represented by $(i - j)$ contribute a lower cost (lines 13–22). The lowest cost from combining two subsets is kept in $minValue$, and the integer that represents the corresponding subset is kept in $bestSplit$. If the $minValue$ contributed by aggregating two subsets is smaller than the current lowest cost of the keyword subset represented by i, we update its cost $\mathsf{Cost}[i]$ and update the group of objects $\mathsf{Group}[i]$ that covers it (lines 23–25). Finally, we return the lowest cost and the best group (line 26).

The algorithm scans the whole dataset D in lines 3–10, and it finds the best group according to Equation 6 in lines 11–22. Therefore, it runs exponentially in terms of the number of query keywords and linearly in terms of the size of the dataset.

Example 3.2: We proceed to illustrate Algorithm 2. Given a query q with the keywords set $q.\psi$ = $\{t_1, t_2, t_3\}$ and three objects o_1, o_2, and o_3 with description: $o_1.\psi$ = $\{t_1, t_2\}$ and $\mathsf{Dist}\ (q, o_1)$ = 1; $o_2.\psi$ =$\{t_2, t_3\}$ and $\mathsf{Dist}\ (q, o_2)$ = 2; $o_3.\psi$ = $\{t_1, t_2, t_3\}$ and $\mathsf{Dist}\ (q, o_3)$ = 4. Lines 3–10 return the following results:

i	1	2	3	4	5	6	7
Cost	1	1	1	2	4	2	4
Group	o_1	o_1	o_1	o_2	o_3	o_2	o_3

We subsequently utilize these values to compute the final results. For example, when computing Cost[5], we determine that $\{o_1, o_2\}$(Cost[1]+Cost[4]) is better than $\{o_3\}$, and we update its value. Finally, we have the following results:

i	1	2	3	4	5	6	7
Cost	1	1	1	2	3	2	3
Group	o_1	o_1	o_1	o_2	o_1, o_2	o_2	o_1, o_2

□

3.3.2 Exact Algorithm Using an Index

The dynamic programming algorithm presented above needs to scan the whole dataset. This has two drawbacks: (1) it wastes computation when checking many unnecessary objects that do not contain any query keyword, and (2) all the objects whose text descriptions overlap with the query keywords are scanned to obtain the lowest costs for the query keyword subsets.

To overcome the first drawback, we utilize the IR-tree that enables us to retrieve only the objects that contain some query keywords while avoiding checking the objects containing no query keywords. For the second drawback, we show that it is not always necessary to scan all the objects covering part of the query keywords.

We propose the following principle for our algorithm: we process objects in ascending order of their distances to a query q. By following that order, we know that the lowest cost of a subset is always contributed by a single or a group of closer objects based on Lemma 3.3.

Lemma 3.4: *Consider a query q. If we process objects in ascending order of their distances to q, when we reach an object o_i containing a query keyword subset ws, all subsets of ws will get their lowest costs.*

Proof: *Obvious since all objects to be visited after o_i have larger cost for any subset of ws; thus, its lowest cost is either contributed by o_i or by objects visited earlier.* □

Example 3.3: Recall the query in Example 3.1. We first process object o_1, and we know 1 is the lowest costs of subsets $\{t_1, t_2\}$, $\{t_1\}$, and $\{t_2\}$. Then we reach o_2, and we know 2 is the lowest costs of subsets $\{t_2, t_3\}$ and $\{t_3\}$ ($\{t_2\}$ already has lowest cost 1). □

Based on Lemma 3.4, we can derive a stopping condition for our algorithm—we reach an object that contains all the query keywords. However, if no such object exists in the dataset, the algorithm is still required to scan to the furthest object containing some query keywords before it can stop. In the example in Table 2, we need to read all the four objects. But if the third furthest object o_3 covers $\{t_1, t_2, t_3\}$, we need not read o_4.

We proceed to present an additional stopping condition.

Lemma 3.5: *Given two query keyword subsets ws_i and ws_j, and with union $ws_u = ws_i \cup ws_j$, we have* $\mathsf{Cost}(ws_u) \leq \mathsf{Cost}(ws_i)$ + $\mathsf{Cost}(ws_j)$.

Proof: *Obvious from Equations 3–5.* □

Based on Lemma 3.5, for any two keyword subsets whose lowest costs are known, we can obtain an upper bound of the lowest cost value for the keyword subset that is the union of the two keyword subsets. We denote the upper bound by $Cost_u$.

In our algorithm, we keep track of the upper bounds for the subsets whose costs are still unknown. Whenever we reach an object from which some keyword subset gets its lowest cost (according to Lemma 3.4), the subset, together with each of the subsets that have either lowest costs or upper bounds of cost (i.e., the keyword subsets that are covered by visited objects), are used to update the upper bound cost values of the corresponding union keyword subsets.

Example 3.4: Recall Example 3.1. After the object o_2 is scanned, $\{t_3\}$ gets its lowest cost 2. We can compute an upper bound for $\{t_1, t_3\}$ using the costs of $\{t_1\}$ and $\{t_3\}$, i.e., $Cost_u(\{t_1, t_3\})$ = $\mathsf{Cost}(\{t_1\})$+$\mathsf{Cost}(\{t_3\})$ = 3 (covered by o_1 and o_2). Similarly, we can also compute an upper bound value 3 for $\{t_1, t_2, t_3\}$ using the costs of $\{t_1, t_2\}$ and $\{t_3\}$, and the set is also covered by o_1 and o_2.

When we reach o_3, we get a lower cost of 2.5 for $\{t_1, t_3\}$(the previous upper bound of 3 is updated). Then $\{t_1, t_3\}$ are combined with $\{t_2\}$ to form $\{t_1, t_2, t_3\}$ with a cost of 3.5. Since this value exceeds its current upper bound, no update is needed. □

We are ready to introduce a lemma that provides an early stopping condition for our algorithm.

Lemma 3.6: *Suppose that we scan objects in ascending order of their distances to q. Given a keyword subset ws, when we reach object o_i, and if* $\mathsf{Dist}(q, o_i) \geq Cost_u(ws)$, *then* $\mathsf{Cost}(ws) = Cost_u(ws)$, *where $Cost_u(ws)$ is the current upper bound of ws.*

Proof: *We prove this by contradiction. If any object o_j further to q than o_i is a member of the best group then it must have*

$$\mathsf{Cost}(ws) \geq \mathsf{Dist}(q, o_j) \geq \mathsf{Dist}(q, o_i) \geq Cost_u(ws)$$

Since $Cost_u(ws)$ cannot be smaller than $\mathsf{Cost}(ws)$, *no further object will be contained in the best group. In addition, $Cost_u(ws)$ is the current minimum cost value, and thus it becomes the lowest cost of ws.* □

Example 3.5: Recall again Example 3.1. By following ascending order of distances, when the algorithm reaches o_4 (Dist (q, o_4) = 4), we can conclude that $Cost_u(\{t_1, t_2, t_3\})$ = 3 is the lowest cost and that the best group is (o_1, o_2). □

The pseudocode is described in Algorithm 3. All the keyword subsets whose lowest costs are already known are stored in the variable $markedSet$, and the subsets that have upper bounds are stored in the variable $valuedSet$. The IR-tree is used for retrieving the next nearest object that covers some query keywords. We use a min-priority queue U to store the IR-tree nodes and objects, where their distances to the query are the keys.

Algorithm 3: Type1ExactWIndex(q, $irTree$)

```
1  markedSet ← ∅, valuedSet ← ∅;
2  n ← |q.ψ|;
3  for i from 1 to 2^n − 1 do Cost[i] ← ∞, Group[i] ← ∅;
4  U ← new min-priority queue;
5  U.Enqueue(irTree.root, 0);
6  while U is not empty do
7      e ← U.Dequeue();
8      ks ← q.ψ ⋂ e.ψ ;
9      if ks ∉ markedSet then
10         if e is a non-leaf node then
11             foreach entry e′ in node e do
12                 if q.ψ ⋂ e′.ψ ≠ ∅ and q.ψ ⋂ e′.ψ ∉ markedSet
                   then U.Enqueue(e′, minDist(q, e′));
13         else if e is a leaf node then
14             foreach object o in leaf node e do
15                 if q.ψ ⋂ o.ψ ≠ ∅ and q.ψ ⋂ o.ψ ∉ markedSet
                   then U.Enqueue(o, Dist(q, o));
16         else                                  // e is an object
17             foreach set S ∈ valuedSet do
18                 i ← MapToInteger(S);
19                 if Cost[i] < Dist (q, e) then
20                     if i = 2^n − 1 then            // Lemma 3.6
21                         return Cost[2^n − 1] and Group[2^n − 1];
22                     valuedSet ← valuedSet \ S;
23                     markedSet ← markedSet ⋃ S;
24             tempSet ← ∅;
25             foreach subset ss ⊆ ks do // Lemma 3.4
26                 if ss ∉ markedSet then
27                     i ← MapToInteger(ss);
28                     markedSet ← markedSet ⋃ ss;
29                     tempSet ← tempSet ⋃ ss;
30                     if ss ∈ valuedSet then
31                         valuedSet ← valuedSet \ ss;
32                     Cost[i] ← Dist (e, q);
33                     Group[i] ← {e};
34                     if j = 2^n − 1 then            // Lemma 3.4
35                         return Cost[2^n − 1] and Group[2^n − 1];
36             foreach set ts ∈ tempSet do
37                 j ← MapToInteger(ts);
38                 for i from 1 to 2^n − 1 do
39                     if Cost[i] = ∞ then continue;
40                     unionKey ← i|j;
41                     if unionKey = i ∧ unionKey = j then
42                         continue;
43                     D ← Cost[i] + Dist (e, q);
44                     if Cost[unionKey] > D then
45                         Cost[unionKey] ← D;
46                         Group[unionKey] ← Group[i] ⋂ {e};
47 return Cost[2^n − 1] and Group[2^n − 1];
```

The priority queue U is initialized to the root node of the IR-tree (line 4). We dequeue an element e from U, and we compute the keyword intersection ks between e and q (lines 7–8). If the keyword subset ks is contained in $markedSet$ (whose lowest costs are known), we do not need to process e according to Lemma 3.3 (line 9). Otherwise, we process e according to its type: 1) If e is a non-leaf index node, we check each of its child nodes, denoted by e', to see whether e' contains a keyword subset of q that is not contained in $markedSet$. If so, e' is inserted into U with its minimum distance to query q as its priority key (lines 10–12). 2) If e is a leaf node, we handle each object in e similarly to how we handle each child node in 1) (lines 13–15). 3) If e is an object, we first utilize its distance to q to move some keyword subsets from $valuedSet$ to $markedSet$. The subsets whose upper bounds are smaller than $\mathsf{Dist}(q, e)$ get their lowest costs (lines 17–23) according to Lemma 3.6. If the query keyword set $q.\psi$ is confirmed to get its lowest cost, the algorithm terminates (line 21). Then for each subset ss of $q.\psi \cap e.\psi$, if its lowest cost is unknown (line 26), the object e constitutes the best group (Lemma 3.4) for ss. Since ss may already be covered by previously visited objects and have an upper bound of its lowest cost, we remove ss from $valuedSet$ (lines 30–31). Once $q.\psi$ gets its lowest cost, the algorithm terminates (lines 34–35). In lines 36–46, we combine each subset ts that newly obtained its lowest cost with the subsets that already have cost values (Lemma 3.5). In line 40, "|" is the bit-wise OR operator. If one is the subset of the other (line 41), we do not combine the two subsets; otherwise, we update the cost value for the union of the two subsets (lines 44–46).

This algorithm runs faster than Type1ExactNoIndex due to two reasons: first, using the IR-tree avoids scanning the whole dataset; second, based on Lemma 3.6, we are able to find the best group without scanning objects whose distances exceed the cost of the current group.

Example 3.6: Recall Table 2 in Example 3.1. The algorithm works as follows. 1) After processing o_1, the result is shown in Table 3, in which i is the integer representing a keyword subset and status "M" means that the subset is contained in $markedSet$. Table 3 shows that $\{t_1\}$ ($i = 1$), $\{t_2\}$ ($i = 2$), and $\{t_1, t_2\}$ ($i = 3$) get their lowest costs and best groups. 2) After processing o_2, the result is shown in Table 4. Except for $\{t_1, t_3\}$ and $\{t_1, t_2, t_3\}$, all the subsets obtain their lowest costs. The cost values of the two subsets are obtained by combining other subsets with known lowest cost. The status value "V" means that the subset is stored in $valuedSet$. 3) After processing o_3, we have the result shown in Table 5. Here, $\{t_1, t_3\}$ gets its lowest cost since it is covered by o_3. 4) When we reach o_4, since its distance to the query is already larger than the currently lowest cost of $\{t_1, t_2, t_3\}$ (the only element in $valuedSet$), we do not need to process it. Set $\{t_1, t_2, t_3\}$ gets the lowest cost value 3 and is moved to $markedSet$. We now find the best group and the lowest cost.

i	1	2	3	4	5	6	7
Cost	1	1	1	∞	∞	∞	∞
Group	o_1	o_1	o_1	null	null	null	null
Status	M	M	M	null	null	null	null

Table 3: Results after processing o_1

i	1	2	3	4	5	6	7
Cost	1	1	1	2	3	2	3
Group	o_1	o_1	o_1	o_2	o_1,o_2	o_2	o_1,o_2
Status	M	M	M	M	V	M	V

Table 4: Results after processing o_2

i	1	2	3	4	5	6	7
Cost	1	1	1	2	2.5	2	3
Group	o_1	o_1	o_1	o_2	o_3	o_2	o_1,o_2
Status	M	M	M	M	M	M	V

Table 5: Results after processing o_3 □

4. PROCESSING TYPE2 SPATIAL GROUP KEYWORD QUERIES

We present two approximation algorithms in Sections 4.1 and 4.2 and an exact algorithm in Section 4.3.

Algorithm 4: Type2Appro1(q, $irTree$)

```
1  U ← new min-priority queue;
2  U.Enqueue(irTree.root, 0);
3  V ← ∅;
4  uSkiSet ← q.ψ ;                        // uncovered keywords
5  while U is not empty do
6      e ← U.Dequeue();
7      if e is an object then
8          V ← V ∪ e ;                     // add e to result
9          uSkiSet ← uSkiSet \ e.ψ;
10         if uSkiSet=∅ then break;
11     else
12         read the posting lists of e for keywords in uSkiSet;
13         foreach entry e′ in node e do
14             if uSkiSet ∩ e′.ψ ≠ ∅ then
15                 if e is a non-leaf node then
16                     U.Enqueue(e′, minDist(q, e′));
17                 else U.Enqueue(o, Dist(q, o));
18 return V;                               // results
```

4.1 Approximation Algorithm 1

Given a query q, the idea of the algorithm, called Type2Appro1, is to find the nearest object for each keyword t_i in $q.\psi$. The set of all such nearest objects make up the result set. The pseudocode, which assumes that the dataset is indexed using the IR-tree, is outlined in Algorithm 4. The algorithm uses a min-priority queue U for the best-first search. In each iteration, we dequeue an element e from U. If e is an object, we push it into the result set and update the uncovered keyword subset (lines 7–10); if e is a node in the IR-tree, we insert all its child nodes that contain some uncovered keywords into U (lines 12–17). The runtime of this algorithm is linear in the number of query keywords.

Example 4.1: Consider a query $q.\psi = \{t_1, t_3, t_5\}$ and the objects shown in Figure 1. The object o_1 covering t_1 is first added to the result. Then o_2 containing t_3 is added, and when o_4 containing t_5 is retrieved, we obtain a group. Object o_4 has the maximum distance to query, which is 4. The maximum diameter is 6, which is the distance between o_2 and o_4. Hence, the cost of this group is 10. □

We proceed to show that Type2Appro1 is within an approximation factor of 3.

Theorem 4.1: *The cost of the solution V, returned by* Type2Appro1 *for a given query q, is at most three times the cost of the optimal solution OPT:* $\mathsf{Cost}(V) \le 3 \cdot \mathsf{Cost}(OPT)$.

Proof: *Let $d = max_{o_i \in V}\{\mathsf{Dist}(o_i, q)\}$, where V is the solution returned by* Type2Appro1. *Obviously, the optimal solution OPT satisfies* $\mathsf{Cost}(OPT) \ge d$.

For the solution V, the largest possible distance between two objects in V is $2d$. Thus, we have the following cost: $\mathsf{Cost}(V) \le d + 2d \le 3 \cdot \mathsf{Cost}(OPT)$. □

4.2 Approximation Algorithm 2

Based on Type2Appro1, we present an algorithm with a better approximation bound.

Let o_f be the furthest object returned by Type2Appro1, and let t_s be the keyword covered by o_f, but not by nearer objects in the result set. We create a new query q_{o_f} using the position of o_f and the keywords of the original query q, i.e., $q_{o_f}.\lambda = o_f.\lambda$ and $q_{o_f}.\psi = q.\psi$. We invoke Type2Appro1 to find a group of objects for q_{o_f}. We compute the cost of this group with respect to q, and this cost serves as the initial lowest cost.

```
Algorithm 5: Type2Appro2( q, irTree)
1  U ← new min-priority queue;
2  U.Enqueue(irTree.root, 0);
3  V ← Type2Appro1(q, irTree);
4  Cost_V ← the cost of V;
5  t_s ← the word only covered by the furthest object in V;
6  while U is not empty do
7      e ← U.Dequeue();
8      if e is not an object then
9          if minDist(q, e) ≥ Cost_V then break;
10         foreach entry e' in node e do
11             if t_s ∈ e'.ψ then
12                 if e is a non-leaf node then
13                     U.Enqueue(e', minDist(q, e'));
14                 else U.Enqueue(e', Dist(q, e'));
15     else
16         if Dist(q, e) ≥ Cost_V then break;
17         q_e.λ ← e.λ; q_e.ψ ← q.ψ;
18         V' ← Type2Appro1(q_e, irTree);
19         Cost_V' ← the cost of V';
20         if Cost_V' < Cost_V then
21             Cost_V ← Cost_V';
22             V ← V';
23 return V
```

Then we incrementally find the next nearest objects containing keyword t_s. For each such object o_{t_s}, we create a query for o_{t_s} in the same way as for o_f. Similarly, Type2Appro1 is used to find a group of objects for the query. If the cost of this group is smaller than $Cost_V$, we update $Cost_V$, and this group becomes the current best group. This process is repeated until we reach an object whose distance to q is larger than $Cost_V$, or until we have considered all objects containing keyword t_s.

The pseudocode is given in Algorithm 5. In lines 3–4, we find a group V that satisfies the query using Type2Appro1, and we compute its cost $Cost_V$. Then we find the word t_s that is only contained in the furthest object in V (line 5). In lines 6–22, we incrementally search for the next nearest objects containing t_s within the range of $Cost_V$ to q. We dequeue an element e from U. If it is an IR-tree node, we check if its minimum distance exceeds $Cost_V$ (line 9). If so, the algorithm terminates since all further-away objects have a higher cost than the current best solution and will not be contained in the result group. Otherwise, we read all its child nodes and insert the nodes that contain t_s into U according to their minimum distances to q (lines 10–14). If e is a spatial object, we also compare its distance to q with $Cost_V$ to determine whether the algorithm terminates (line 16). In lines 17–19, we create a new query q_e with the position of e and the texts of q, and we then find a group using q_e as the query and compute its cost. If this new cost is smaller than $Cost_V$, we update $Cost_V$ and the current best group (lines 20–22). Finally, we return V as the result group.

Since we invoke algorithm Type2Appro1 on each object containing t_s, in the worst case, this algorithm runs linearly in terms of both the number of query keywords and the size of the dataset. However, in practice, only a fraction of the dataset needs to be considered.

Example 4.2: Recall query q and the dataset in Example 4.1. Algorithm Type2Appro1 is invoked to return a group $\{o_1, o_2, o_4\}$, in which o_4 is the furthest and contains t_5. We search for a group near o_4, that is $\{o_1, o_3, o_4\}$ with cost 9 ($\mathsf{Dist}(q, o_4)$ + $\mathsf{Dist}(o_3, o_4)$ = 4 + 5). The next nearest object containing t_5 is o_7. For o_7, we find a group $\{o_2, o_7, o_8\}$ with cost 8 ($\mathsf{Dist}(q, o_8)$ + $\mathsf{Dist}(o_7, o_8)$ = 6 + 2), which is better than that of the previous one. Therefore $\{o_2, o_7, o_8\}$ is the result. Note that the optimal group is $\{o_1, o_2, o_7\}$ with cost 7.5 (($\mathsf{Dist}(q, o_7)$ + $\mathsf{Dist}(o_7, o_1)$ = 5 + 2.5). □

We proceed to study the approximation ratio of the algorithm.

Lemma 4.2: *Given an object o_j containing t_s, the cost of the group found at the position of o_j, denoted by $Cost_V(o_j)$, is in the following range: $\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max}) \le Cost_V(o_j) \le \mathsf{Dist}(q, o_j) + 3\mathsf{Dist}(o_j, o_{max})$, in which o_{max} is the furthest object from o_j in the group.*

Proof: *1) $\mathsf{Dist}(q, o_j)$ is a lower bound on the distance of this group to q , and $\mathsf{Dist}(o_j, o_{max})$ is a lower bound on the diameter of this group. As a result, the minimum cost of this group is $\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max})$.*

2) $\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max})$ is the maximum possible distance to q of this group. Further, the diameter will not exceed $2\mathsf{Dist}(o_j, o_{max})$ since every object of this group is in the circle with center o_j and radius $\mathsf{Dist}(o_j, o_{max})$. Therefore, the cost of this group is upper bounded by $\mathsf{Dist}(q, o_j) + 3\mathsf{Dist}(o_j, o_{max})$.

□

Lemma 4.3: *Let OPT denote the optimal group, and let o_j denote the object containing word t_s in OPT, and let o_{max} denote the furthest object to o_j in group V found by* Type2Appro1. *It holds that: $Cost_{OPT} \ge \mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max})$.*

Proof: *Object o_{max} must contain some keyword, denoted by t, which is not covered by the other objects in V. Among objects containing t, o_{max} is the closest to o_j. Therefore, in OPT, the object covering t cannot be closer to o_j than o_{max}. As a result, $\mathsf{Dist}(o_j, o_{max})$ is a lower bound on the diameter of OPT. $\mathsf{Dist}(q, o_j)$ is a lower bound on the distance between q and OPT. Therefore, $Cost_{OPT} \ge \mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max})$.* □

Lemma 4.4: *The approximation ratio of algorithm* Type2Appro2 *is not larger than 2.*

Proof:*Let o_j denote the object containing word t_s in the optimal group OPT. Let $Cost_{APPR}$ be the cost returned by Algorithm* Type2Appro2. *We know that $Cost_{APPR} \le Cost_V(o_j)$, because $Cost_{APPR}$ is the smallest among all cost values of groups found at each object containing t_s. Let o_i be the nearest object containing t_s. We get $Cost_{APPR} \le Cost_V(o_i) \le 3\mathsf{Dist}(q, o_i)$ according to Theorem 4.1. Therefore,*

$$\frac{Cost_{APPR}}{Cost_{OPT}} \le \frac{3\mathsf{Dist}(q, o_i)}{Cost_{OPT}} \le \frac{3\mathsf{Dist}(q, o_i)}{\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max})}$$
$$= 2 + \frac{3\mathsf{Dist}(q, o_i) - 2(\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max}))}{\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max})}$$

a) If $\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max}) \ge 1.5\mathsf{Dist}(q, o_i)$ then $Cost_{APPR} Cost_{OPT} \le 2$.
b) If $\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max}) < 1.5\mathsf{Dist}(q, o_i)$, because $\mathsf{Dist}(q, o_j) \ge \mathsf{Dist}(q, o_i)$, then $\mathsf{Dist}(o_j, o_{max}) \le 0.5\mathsf{Dist}(q, o_j)$.
Since $Cost_{APPR} \le Cost_V(o_j)$, we have:

$$\frac{Cost_{APPR}}{Cost_{OPT}} \le \frac{Cost_V(o_j)}{Cost_{OPT}}$$
$$\le \frac{\mathsf{Dist}(q, o_j) + 3\mathsf{Dist}(o_j, o_{max})}{\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max})}$$
$$= 2 + \frac{\mathsf{Dist}(o_j, o_{max}) - \mathsf{Dist}(q, o_j)}{\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max})}$$
$$\le 2 - 0.5\frac{\mathsf{Dist}(o_j, o_{max})}{\mathsf{Dist}(q, o_j) + \mathsf{Dist}(o_j, o_{max})} \le 2$$

Thus, we complete the proof. □

4.3 Exact Algorithm

It is challenging to develop an exact algorithm for TYPE2 spatial group keyword queries, as it appears that an exact algorithm cannot avoid an exhaustive search of the object space.

We utilize the Type2Appro2 algorithm to first derive an upper bound cost for the best group and then use this cost to bound the exhaustive search in the object space. Specifically, we develop several pruning strategies to prune the enumeration space. With these efforts, we expect the exact algorithm to be reasonably efficient when the dataset contains at most tens of thousands of objects and the number of query keywords is small.

Before presenting the idea underlying the algorithm, we define the concept of *covering node set* and the lower bound cost of such a set.

DEFINITION 1 (COVERING NODE SET). *Given a query q, a covering node set is a set of nodes that cover the query keywords, with each node contributing at least one object to the final result.*

DEFINITION 2. *Given a query q and a covering node set N,*
1) if N contains only one node e, its lower bound cost is:

$$minCost(N) = \mathsf{minDist}(q, e)$$

2) if N contains multiple nodes, its lower bound cost is:

$$minCost(N) = max_{e_i \in N}\mathsf{minDist}(q, e_i) + max_{e_j, e_k \in N}\mathsf{minDist}(e_j, e_k),$$

where $max_{e_i \in N}\mathsf{minDist}(q, e_i)$ is the minimum distance from q to a group from N, and $max_{e_j, e_k \in N}\mathsf{minDist}(e_j, e_k)$ is the minimum diameter of a group from N. Therefore, $minCost(N)$ is the lower bound of the cost of the best group from N.

The algorithm's idea is to perform a best-first search on the IR-tree to find the covering node sets, with some objects from these nodes constituting a group satisfying the keywords requirement of a query. We process the covering node set with the lowest cost to find covering node sets from their child nodes. When we reach a covering node set consisting of leaf nodes, we find a group of objects with the lowest cost by performing an exhaustive search.

The pseudocode is given in Algorithm 6. A priority queue U stores the covering node sets. Algorithm 4 (Type2Appro1) is invoked to find a group, and its cost serves as the current lowest cost (lines 3–4). We next search from the root node, enumerating all its child node sets to find covering node sets. If a node set covers the query keywords, we estimate its lower bound cost by Definition 2 and insert it into U with the estimated cost as the key. After we finish enumerating the covering node sets of a covering node set, we dequeue a covering node set N from U, and we find its lower level covering node sets that cover the query keywords. The covering node sets whose lower bounds are smaller than the current lowest cost are inserted into U (lines 14–17). Once we reach a leaf node, we do an exhaustive search to get the best group in the covering node set, and we update the lowest cost stored in $Cost_V$ with the cost of this group (lines 10–12).

The algorithm terminates when the lower bound of the covering node set at the top position of U is larger than the current lowest cost (line 7) because the remaining covering node sets in U have larger costs than the current lowest cost, and because covering node sets at their lower levels do not contain better groups according to Lemma 4.5.

Lemma 4.5: *For any lower level covering node set L enumerated from covering node set N, we have $minCost(L) \geq minCost(N)$.*

Algorithm 6: Top-Down Search $(q, irTree)$

1 $U \leftarrow$ new min-priority queue;
2 U.Enqueue($irTree.root$,0);
3 $V \leftarrow$ Type2App2(q, $irTree$);
4 $Cost_V \leftarrow$ the cost of V;
5 **while** *U is not empty* **do**
6     $N \leftarrow U$.Dequeue();
7   **if** *minCost(N)* $\geq Cost_V$ **then break**;
8   **if** *N contains leaf nodes* **then**
9         $V' \leftarrow$ ExhaustiveSearch(N);
10         $Cost_{V'} \leftarrow$ the cost of V';
11   **if** $Cost_{V'} < Cost_V$ **then**
12             $Cost_V \leftarrow Cost_{V'}$; $V \leftarrow V'$;
13   **else**
14         $S \leftarrow$ EnumerateNodeSets(N, $Cost_V$, q);
15   **foreach** *node set ns in S* **do**
16   **if** $q.\psi \subseteq ns.\psi$ **then**
17   U.Enqueue(ns, minCost(ns));
18 **return** V and $Cost_V$

Proof: *a) Denote by l_a the child node of the node n_i that is the furthest from Q in N. We have:* $\mathsf{minDist}(q, l_a) \geq \mathsf{minDist}(q, n_i)$. *b) Denote by l_b and l_c the child nodes of n_j and n_k that have the largest distance among all pairs of child nodes from n_j and n_k. We then have:* $\mathsf{minDist}(l_b, l_c) \geq \mathsf{minDist}(n_j, n_k)$. *In addition, as l_a, l_b, and $l_c \in L$, it is true that* $max_{l_i \in L}\mathsf{minDist}(q, l_i) \geq \mathsf{minDist}(q, l_a)$*, and* $max_{l_j, l_k \in L}\mathsf{minDist}(l_j, l_k) \geq \mathsf{minDist}(l_b, l_c)$. *We then obtain* $minCost(L) \geq \mathsf{minDist}(q, l_a) + \mathsf{minDist}(l_b, l_c) \geq minCost(N)$. □

Algorithm 7: EnumerateNodeSets (N, $Cost$, q)

1 $setList \leftarrow \emptyset$;
2 **foreach** *node n_i in N* **do**
3     $cList_i \leftarrow \emptyset$;
4     $L_i^1 \leftarrow \emptyset$;
5   **foreach** *child node c_i of n_i* **do**
6   **if** *minCost(c_i)* $\geq Cost$ **then** $L_i^1 \leftarrow L_i^1 \bigcup c_i$;
7   **for** *m from* 2 *to* $(|q.\psi| - |N| + 1)$ **do**
8         $L_i^m \leftarrow \emptyset$;
9   **foreach** *node set* $NS_1 \in L_i^{m-1}$ **do**
10   **foreach** *node set* $NS_2 \in L_i^{m-1}$ **do**
11   **if** *NS_1 and NS_2 share the first $(m-1)$ nodes* **then**
12                     $NS \leftarrow$ Merge(NS_1, NS_2);
13   **if** *minCost(NS)* $< Cost$ **then**
14                         $L_i^m \leftarrow L_i^m \bigcup NS$;
15         $cList_i \leftarrow cList_i \bigcup L_i^m$;
16 **foreach** *node set ns formed by node sets selected from each of $cList_1 ... cList_n$* **do**
17   **if** *minCost(ns)* $< Cost$ **then**
18         $setList \leftarrow setList \bigcup ns$;
19 **return** *setList*;

We proceed to describe EnumerateNodeSets called in line 14, which enumerates all possible lower level covering node sets from an upper level covering node set. First, we consider the simple case where the covering node set N only contains one node. We use a bottom-up method to find all the covering node sets. Let $n = |q.\psi|$ be the number of keywords in query q. The size of a covering node set is at most n based on the pigeon-hole principle.

We first enumerate the covering node sets that consist of a single node. Then these covering node sets are combined to form covering node sets with two members. In general, all the covering node sets of size m can be combined by two covering node sets of size $(m-1)$. If two covering node sets with size $(m-1)$ share the first $(m-2)$ nodes and the lower bound cost of the new combined node set is smaller than that of the current lowest cost, this new set is a candidate covering node set.

Next, we move to the case where a covering node set N contains more than one node. For each node, we follow the previous method to get a list of candidate node sets. However, we do not need to enumerate all combinations. Rather, we only need to enumerate its child nodes up to size $(|q.\psi|-|N|+1)$ according to the pigeon-hole principle (each node in N must contribute at least one child node). After we get a list of child node sets from each node, we select a node set from each list and merge them to determine whether they are a covering node set and to learn whether the merged node set has a lower bound cost that is smaller than the current lowest cost. The details of EnumerateNodeSets are covered in Algorithm 7.

5. EXPERIMENTAL STUDY

5.1 Experimental Settings

Algorithms. For the TYPE1 spatial group keyword query, we consider the approximation algorithm from Section 3.1 (denoted by T1A1), the exact algorithm without index from Section 3.2 (denoted by T1E1), and the exact algorithm utilizing the IR-tree from Section 3.3 (denoted by T1E2). For the TYPE2 spatial group keyword query, we study the two approximation algorithms in Sections 4.1 and 4.2 (denoted by T2A1 and T2A2, respectively) and the exact algorithm from Section 4.3 (denoted by T2E1).

Dataset and queries. We use three datasets. Table 6 shows some properties of these datasets. Dataset GN is extracted from the U.S. Board on Geographic Names (geonames.usgs.gov), in which each object is a location with a geographic name (e.g., valley). Dataset Web is generated from two real datasets. One is WEBSPAM-UK2007[1] that consists of a large number of web documents; the other is a spatial dataset containing the tiger Census blocks in Iowa, Kansas, Missouri, and Nebraska (www. rtreeportal.org). We randomly combine web documents and spatial objects to get the Web dataset. Dataset Hotel contains spatial objects that represent some hotels in the U.S. (www.allstays.com). Each object has a location and a set of words that describe the hotel (e.g., restaurant, pool).

Hotel is small and is used to evaluate the performance of our algorithms when the dataset and index are memory resident, and the other two large datasets are used to evaluate our algorithms when the dataset and index are disk-based.

Property	Web	GN	Hotel
Number of objects	579,727	1,868,821	20,790
Number of unique words	2,899,175	222,409	602
Number of words	249,132,883	18,374,228	80,845

Table 6: Dataset properties

We generate 5 query sets in the space of GN, in which the number of keywords is 3, 6, 9, 12, and 15, respectively. We also generate 5 similar query sets in the space of both Web and Hotel. Each set comprises 50 queries. We rank words in descending order of their frequencies in each dataset. The keywords of each query are randomly generated from the words in the percentile range of 10–40, and the location is randomly generated in the whole region of the dataset. Such queries would need similar processing time, and we report the average cost of queries in each query set. We also conduct experiments on queries generated in other ways, and a summary of experimental results is presented in Section 5.2.3.

Setup. The IR-tree index structure is disk resident, and the page size is 4KB. The number of children of a node in the IR-tree is computed given the fact that each node occupies a page. This translates to 100 children per node in our implementation.

All algorithms were implemented in VC++ and run on an Intel(R) Core(TM)2 Duo CPU E6550 @2.66GHz with 4GB RAM.

5.2 Experimental Results

5.2.1 *Type 1 Spatial Group Keyword Query*

Efficiency of different algorithms. The objective of this set of experiments is to study the efficiency of the three algorithms when we vary the number of query keywords. Figure 3 shows the runtime of the three algorithms on the dataset GN. As expected, the approximation algorithm T1A1 runs much faster than the two exact algorithms, i.e., T1E1 and T1E2. The runtime of the approximation algorithm T1A1 increases almost linearly with the number of query keywords. It is understandable that its running time is in proportion to the number of query keywords: T1A1 keeps searching for the object with the lowest cost that covers part or all of the query keywords, and it terminates when a group of objects that covers the query keywords has been found.

For the two exact algorithms, T1E1 runs slower than T1E2. T1E1 needs to scan the whole dataset and process all the objects that overlap with the query keywords. In contrast, T1E2 avoids scanning objects that do not contain query keywords by utilizing the IR-tree and can avoid accessing the objects whose distances to the query are larger than the cost of the discovered group, thus pruning the search space significantly. The experimental results demonstrate the usefulness of the IR-tree based pruning strategies. It can also be observed that the runtimes of T1E1 and T1E2 increase with the number of keywords; however, the increase is not exponential. The reason is that computing the costs of objects dominates the running time over the dynamic programming component (with an exponential complexity in terms of the number of keywords).

The runtime is consistent with I/O costs, and we do not report I/O costs due to the space limitation.

Accuracy of the approximation algorithm. Figure 4 shows the accuracy of T1A1 on GN. The approximation algorithm is capable of achieving very accurate results.

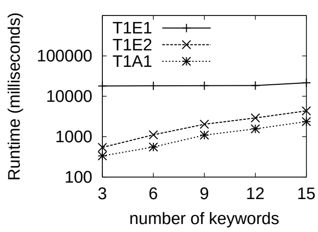


Figure 3: Runtime (GN)

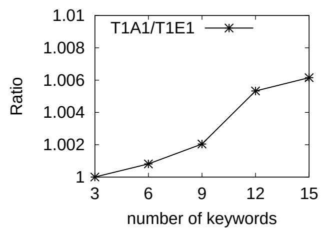


Figure 4: Appro. Ratio (GN)

Experiments on Web. This experiment studies the efficiency and accuracy on the dataset Web in which each object is associated with a large set of keywords. Figure 5 shows the runtime of the algorithms T1A1 and T1E2, and Figure 6 shows the accuracy of T1E2. To ensure readability of the figures, we omit T1E1 since it is inferior to T1E2 (in orders of magnitude). We observe qualitatively similar results on Web as we do on GN.

Experiments on the memory-resident dataset Hotel. This experiment studies the performance of our algorithms when the dataset

[1]http://barcelona.research.yahoo.net/webspam/datasets/uk2007

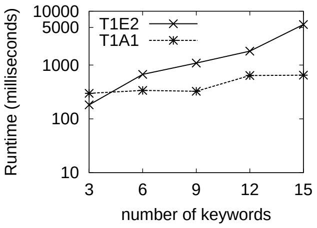


Figure 5: Runtime (Web)

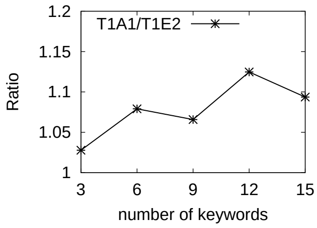


Figure 6: Appro. Ratio (Web)

and index are in memory. Figure 7 shows the runtime and Figure 8 shows the accuracy of T1A1. We observe qualitatively similar results as we do on the two disk-resident datasets. T1E2 takes only slightly more time than T1A1 when the number of keywords is smaller than 12, while T1E1 performs much slower. We also conduct experiments on Hotel when the data and index are disk-based, and we observe qualitatively similar results.

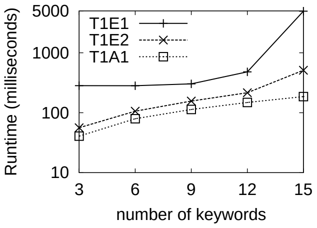


Figure 7: Runtime (Hotel)

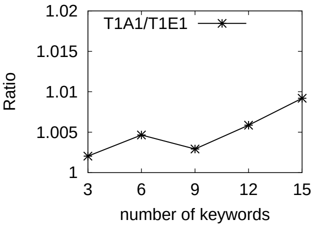


Figure 8: Appro. Ratio(Hotel)

Scalability. To evaluate scalability, we generate 5 datasets containing from 2 to 10 million objects: we generate new locations by copying the locations in GN to nearby locations while maintaining the real distribution of the objects; for each new location, a document is selected randomly from the text descriptions of the objects in GN. Figure 9 shows the runtime of T1A1 and T1E2 (the number of query keywords is 6). T1A1 scales well with the size of the dataset. The exact algorithm T1E2 also scales well. The accuracy changes only slightly; we omit the details due to the space limitation. T1E1 runs much slower than T1A1 and T1E2 and is omitted.

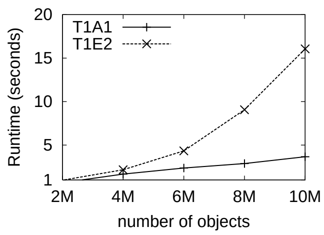


Figure 9: Scalability (Type1)

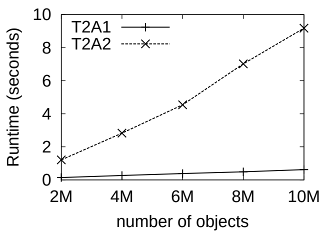


Figure 10: Scalability (Type2)

5.2.2 *Type 2 Spatial Group Keyword Query*

Efficiency of the algorithms and accuracy of the approximation algorithms. The objective of this set of experiments is to evaluate the efficiency of the two approximation algorithms T2A1 and T2A2 and the exact algorithm T2E1, and to evaluate the accuracy of the two approximation algorithms, when we vary the number of query keywords. Figure 11 shows the runtime of the three algorithms, and Figure 12 shows the accuracy of T2A1 and T2A2 on GN.

As can be seen, T2A1 outperforms T2A2 in terms of runtime, and the accuracy of T2A1 is worse than that of T2A2. This is because T2A1 terminates once a group of nearest objects covering query keywords is found. In contrast, T2A2 may invoke T2A1 multiple times. Both T2A1 and T2A2 achieve good accuracy compared with the optimal group returned by T2E1. T2E1 is able to find the optimal group. However, it is much slower than the two approximation algorithms. As expected, with the increase in the number of keywords, the runtime of T2E1 increases exponentially due to its enumerating the covering node sets and performing exhaustive search. When the number of keywords is small, its runtime would be reasonable for applications without a high demand on query time, e.g., finding research partners. However, approximation algorithms represent a better option when the query time is essential.

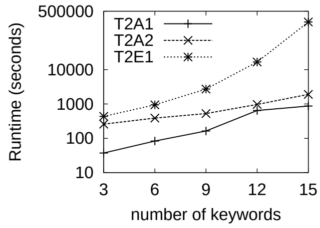


Figure 11: Runtime (GN)

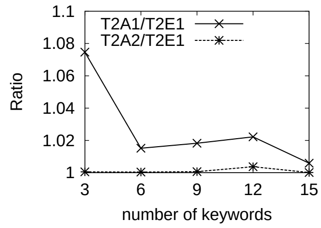


Figure 12: Appro. Ratio (GN)

Experiments on Web. Figure 13 shows the runtimes of T2A1, T2A2, and T2E1. Figure 14 shows the accuracy of T2A1 and T2A2. We observe qualitatively similar results on Web as we do on GN.

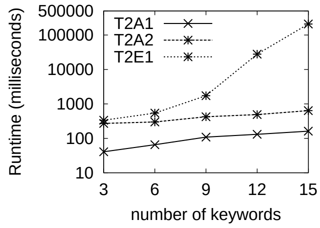


Figure 13: Runtime (Web)

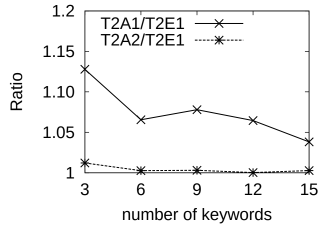


Figure 14: Appro. Ratio(Web)

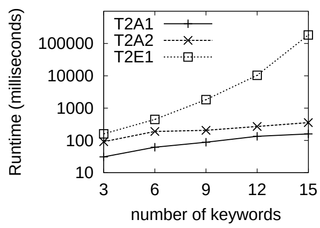


Figure 15: Runtime (Hotel)

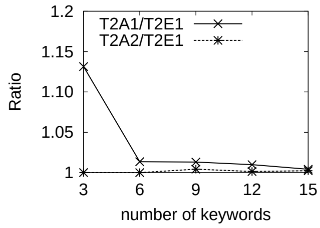


Figure 16: Appro. Ratio(Hotel)

Experiments on memory-resident dataset Hotel. We evaluate performance when the dataset and index are in memory using Hotel. Figure 15 shows the runtime of the three algorithms on Hotel, and Figure 16 shows the accuracy of T2A1 and T2A2. The results are consistent with those obtained for disk-resident datasets. T2A2 runs slower than T2A1 while it has better accuracy. The runtime of T2E1 increases exponentially with the number of keywords. When the dataset Hotel and index are disk-based, we observe qualitatively similar results.

Scalability. We use the same settings as those used in the scalability experiment for TYPE1 query. Figure 10 shows the runtime of T2A1 and T2A2. Both approximation algorithms scale well with the size of the dataset. The runtime of T2E1 increases exponentially with the dataset size. Thus, to ensure readability of the figure, we omit T2E1, which is orders of magnitudes slower than the approximation algorithms.

5.2.3 Additional Experiments

We also generate queries using the keywords beyond the percentile range of 10–40, and we report a summary of experimental results. From the perspective of efficiency, the approximation algorithms perform similarly on all the queries, irrespective of the frequencies of the query keywords. However, on queries containing frequent words, the runtime of exact algorithm T1E2 is slightly slower than the runtime reported in Section 5.2.1, and T2E1 performs much worse than the results reported in Section 5.2.2. If we use infrequent words as query keywords, the exact algorithms run faster, but are still much slower than the approximate algorithms.

In terms of accuracy, the approximation algorithms have better approximation ratio on queries containing many infrequent words. In the extreme case, when all query keywords occur once in the dataset, the approximation algorithms return the correct results. Generally, the approximation ratio becomes worse when frequent words are used, e.g., around 1.2–1.5 for T1E1.

6. RELATED WORK

Spatial web objects are gaining in prevalence, and numerous works on geographical retrieval study the problem of extracting geographic information from web pages (e.g., [1, 11, 13]), which yields spatial web objects that can subsequently be queried.

Commercial services such as Google and Yahoo! offer local-search functionality. Given a spatial keyword query, they return spatial web objects, e.g., stores and restaurants, near the query location. The results consist of single objects that each satisfy the query in isolation. In contrast, we aim to find groups of objects such that the objects in a group collectively satisfy a query.

Several recently proposed hybrid indexes [5, 8, 10, 14, 15] that tightly integrate spatial indexing (e.g., the R-tree) and text indexing (e.g., inverted lists). In these indexes, each entry e in a tree node stores a *keyword summary field* that concisely summarizes the keywords in the subtree rooted at e. This enables irrelevant entries to be pruned during query processing.

The *IR^2-tree* and the *bR*-tree* [14] augment the R-tree with signatures and bitmaps, respectively. We use the IR-tree [8], covered in Section 3.1, as our index structure due to two features of the IR-tree. First, the fanout of the tree is independent of the number of words of objects in the dataset. Second, during query processing, only (a few) posting lists relevant to the query keywords need to be fetched. However, we note that our proposed algorithms are not tied to the IR-tree, but can be used also with the other tightly combined index.

Most existing works on spatial keyword queries retrieve single objects that are close to the query point and are relevant to the query keywords. In contrast, we retrieve groups of objects that are close to the query point and collectively meet the keywords requirement.

To the best of our knowledge, the only work that retrieves groups of spatial keyword objects relates to the mCK query [14, 15] that takes a set of m keyword as argument. It returns m objects of minimum diameter that match the m keywords. It is assumed that each object in the result corresponds to a unique query keyword. In contrast, our query takes both a spatial location and a set of keywords as arguments, and its semantics are quite different from those of the mCK query.

7. CONCLUSIONS AND FUTURE WORK

We present the new problem of retrieving a group of spatial objects, each associated with a set of keywords, such that the group covers the query's keywords and has the lowest cost measured by their distance to the query point, and the distances between the objects in the group. We study two particular instances of the problem, both of which are NP-complete. We develop approximation algorithms with provable approximation bounds and exact algorithms to solve the two problems. Results of experimental evaluation offer insight into the efficiency and the accuracy of the approximation algorithms, and the efficiency of the exact algorithms.

This work opens to a number of promising directions for future work. First, it is worth extending the algorithm to find top-k groups. An open problem here is to what extent we should allow the overlap among top-k groups. The top-k groups would provide users more options. It might not be useful to return groups sharing lots of members, while it seems to be reasonable to allow a certain degree of overlap groups. Second, it is of interest to develop algorithms for alternative cost functions, such as SUM for both $C_1(\cdot)$ and $C_2(\cdot)$. For the top-k groups problem, one meaningful research issue is to determine which types of cost function are most amenable to efficient processing. Third, it is of interest to consider the problem of partial coverage of query keywords. Finally, treating all query keywords equally may not be suitable in some application scenarios. It is of interest to consider an information retrieval ranking model, such as the vector space model, when computing the text relevance for the spatial group query problem.

8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. C. S. Jensen is an adjunct professor at University of Agder, Norway. This research was supported in part by the Geocrowd project.

9. REFERENCES

[1] E. Amitay, N. Har'El, R. Sivan, and A. Soffer. Web-a-where: geotagging web content. In *SIGIR*, pp. 273–280, 2004.

[2] E. M. Arkin and R. Hassin. Minimum-diameter covering problems. *Networks*, 36(3):147–155, 2000.

[3] A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, 2009.

[4] N. Bourgeois, B. Escoffier, and V. T. Paschos. Efficient approximation of min set cover by moderately exponential algorithms. *Theor. Comput. Sci.*, 410(21-23):2184–2195, 2009.

[5] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.

[6] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pp. 277–288, 2006.

[7] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.

[8] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[9] M. Cygan, L. Kowalik, and M. Wykurz. Exponential-time approximation of weighted set cover. *Inf. Process. Lett.*, 109(16):957–961, 2009.

[10] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pp. 656–665, 2008.

[11] J. Ding, L. Gravano, and N. Shivakumar. Computing geographical scopes of web resources. In *VLDB*, 2000.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pp. 47–57. ACM Press, 1984.

[13] K. S. McCurley. Geospatial mapping and navigation of the web. In *WWW*, pp. 221–229, 2001.

[14] 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*, pp. 688–699, 2009.

[15] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, pp. 521–532, 2010.

[16] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.