Level-aware Collective Spatial Keyword Queries

Pengfei Zhang, Huaizhong Lin, Dongming Lu

College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China

Abstract

The collective spatial keyword query (CoSKQ), which takes a location and a set of keywords as arguments, finds a group of objects that collectively satisfy the user's needs (expressed by keywords) while minimizing the cost function. However, few studies concern the *keyword level* (e.g., the level of attractions), which is of critical importance for decision support in real applications. Motivated by this, we study a novel query paradigm, namely *Level-aware Collective Spatial Keyword* (*LCSK*) *Query*. In our settings, each keyword associated with the objects is tagged with a level. Specifically, the LCSK query asks for a group of objects that cover the query keywords collectively with a threshold constraint and have the minimum cost distance.

We prove the LCSK query is NP-hard, and answer it exactly as well as approximately. The proposed exact algorithm, namely MergeList, answers the query by searching the candidate space progressively with some pruning strategies, which is based on the keyword hash table (KHT) index structure. Unfortunately, this approach is not scalable to large datasets. We thus propose an approximate algorithm called MaxMargin. MaxMargin finds the answer by traversing the proposed LIR-tree using the best-first strategy. Moreover, two pruning strategies are used to improve the query performance. The experiments on real and synthetic datasets verify that the proposed approximate algorithm runs much faster than the competitor with desired accuracy.

Keywords: Collective spatial keyword query, Keyword level, Branch and bound strategy, Triggered update strategy

1. Introduction

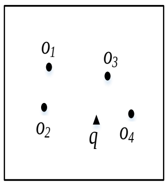
Spatial database has been studied for decades as it supports many applications from people's daily life to scientific research [1]-[10]. Recently, the keyword search has been combined with spatial queries to enhance location-based services such as Baidu Lvyou and Google Earth. Previous works on spatial keyword queries can be roughly classified into two categories based on the answer granularity: (1) some proposals find individual object. Typically, given a location and a set of keywords as arguments, this type of query [? ? ? ?] retrieves individual object that each covers all query keywords, and (2) others ask for a group of objects. In a wide spectrum of applications, whereas multiple objects are required to cover query keywords collectively. Toward this goal, mCK [? ?], CoSKQ [? ?], BKC [?] and SGK [?] are investigated recently. To the best of our knowledge, few studies concern the *keyword level* (e.g., the level of attractions, hotels or personal skills), which is increasingly important for users to make decisions in real applications.

In this work, we explore a novel query paradigm called level-aware spatial keyword (LCSK) query. We enhance CoSKQ from the following two aspects. First, we consider the *level vector* of objects in the underlying database O. We denote by $o.\omega$ the associated keywords with an object $o \in O$. Specifically, the level vector of $o \in O$ is an integer vector, denoted by $o.\nu$. The ith element of a *level vector*, i.e.,

*Corresponding author.

[1]This manuscript is the authors' original work and has not been published nor has it been submitted simultaneously elsewhere.

[2]All authors have checked the manuscript and have agreed to the submission.



object	keywords	level vector	cost	distance
o_1	t_1, t_2, t_3	4, 5, 1	0.1	10
o_2	t_1, t_2, t_4	5, 4, 1	0.1	8
o_3	t_1, t_2, t_3, t_4	3, 2, 5, 5	0.15	6
o_4	t_1, t_2, t_3, t_5	2, 3, 5, 5	0.15	7

Fig. 1. Example of the LCSK query

$o.\nu^i$, represents the level of ith keyword in $o.\omega$, i.e., $o.\omega^i$. Second, we introduce the *weight vector* into the query definition for capturing the user-specified weights assigned to different levels. We consider the cost distance in [**?**] as the cost measurement. The LCSK query has numerous real applications such as resource scheduling, emergency rescue. Next, we present an example of the emergency rescue task.

Example 1. As illustrated in Fig. **??**, we assume that the query point q indicates an earthquake point. There are four rescue teams o_1, o_2, o_3 and o_4 nearby with necessary rescue equipments, e.g., t_1. The *level vector* captures the level of equipments associated with objects, which can be used to measure the rescue ability. The *cost* denotes the overhead of performing the rescue by the corresponding team, and *distance* denotes the Euclidean distance between q and an object.

In such a scenario, multiple teams are necessitated to perform rescue collectively. Next, we show how to retrieve these teams using the LCSK query. We can issue a query $q = (\ell, \omega, W, \theta)$, where ℓ denotes the location of q and $\omega = (t_1, t_2)$ captures the necessary rescue equipments. The normalized weight vector $W = (0.1, 0.15, 0.2, 0.25, 0.3)$ indicates the rescue ability of different levels. As an example, the rescue ability of t_1 in o_1 is 0.25 because the corresponding level is 4. Then, $\theta = 0.5$ denotes the desired rescue ability. That is, a group of teams whose rescue ability are not less than 0.5 are required. There is a tension between response time and rescue ability, thus we enable users to favor one side of the trade-off or the other by adjusting the threshold.

While the group $\{o_3, o_4\}$ outperforms $\{o_1, o_2\}$ in terms of the distance to q, we deliver $\{o_1, o_2\}$ as the answer. This is because : (1) $\{o_3, o_4\}$ fails to achieve the desired rescue ability 0.5 with respect to the required rescue equipments, and (2) we consider the cost distance in [**?**] as the measurement, with which $\{o_1, o_2\}$ has the minimum cost while achieving the desired rescue ability.

More formally, given a spatial database O, and an LCSK query $q = (\ell, \omega, W, \theta)$, where ℓ represents the location and ω is the query keywords. W is a user-specified weight vector and θ is a threshold. The LCSK query is to retrieve a group G of objects such that satisfy the following two conditions:

- $\forall t \in q.\omega$, the sum of the coverage weight of t by G is not less than $q.\theta$;
- The cost distance of G is minimized.

We prove the LCSK query is NP-hard by reduction from the weighted set cover (WSC) problem. We propose two algorithms to answering the query exactly and approximately. The proposed exact algorithm, namely MergeList, performs the query by searching the candidate space progressively, which is based on the flat index structure keyword hash table. Specifically, the candidate space contains all promising answers. We use some strategies to prune the candidate space. Though equipped with some pruning strategies, MergeList is not scalable to large datasets due to the inherent complexity of our problem. We thus develop an approximate algorithm called MaxMargin. MaxMargin finds the answer by traversing the LIR-tree in the best-first fashion. Specifically, LIR-tree augments each inverted file of IR-tree with additional information, i.e., the level of keywords and the cost of objects. Two effective pruning strategies, namely the *branch and bound strategy* and the *triggered update strategy* are proposed to further improve the performance of MaxMargin.

To summarize, we make the following contributions:

- We formally define the LCSK query, which is to find a group of objects such that collectively cover the query keywords with a threshold constraint and have the minimum cost distance. We then theoretically prove this problem is NP-hard.

- We develop an exact algorithm called MergeList with the flat index structure KHT. Furthermore, we propose the hierarchical index structure LIR-tree, which is extended from IR-tree. Based on the LIR-tree, we devise an approximate algorithm called MaxMargin that finds the answer using the best-first strategy. To further facilitate the query processing, two strategies, namely the *branch and bound strategy* and the *triggered update strategy*, are adopted by MaxMargin.
- We conduct comprehensive experiments on real and synthetic datasets to validate the performance of our proposed algorithms.

The rest of this paper is organized as follows: Section 2 reviews the related work. Section 3 formally defines the problem and proves the LCSK query is NP-hard. We elaborate the exact algorithm MergeList in Section 4 and the approximate algorithm MaxMargin in Section 5, respectively. Section 6 gives the empirical study and this paper is concluded in Section 7.

2. Related Work

2.1. Conventional Spatial Keyword Queries

The conventional spatial keyword queries [? ?] take a location and a set of keywords as arguments, and ask for the objects that cover all the query keywords. There are lots of efforts on the conventional spatial keyword queries. We review them as follows.

Combining with top-k queries. The top-k spatial keyword queries return k objects with the highest ranking scores measured by a ranking function, which considers the spatial proximity and the textual relevance. To answer the top-k spatial keyword queries efficiently, various hybrid index structures are investigated. Specifically, this branch includes [? ?] (IR-tree), [?] (SKI), [? ?] (S2I), [?] (IL-Quadtree). Cong et al. [?] use the B^{ck}-tree to facilitate the query processing for the trajectory data. Wu et al. [?] handle the joint top-k spatial keyword queries with the W-IR-Tree, which utilizes keyword partitioning and inverted bitmaps to organize the objects. Zhang et al. [?] show that I^3 index outperforms IR-tree and S2I. Specifically, I^3 adopts the quadtree to hierarchically partition the data space into cells. Gao et al. [?] study the reverse top-k boolean spatial keyword queries using the count tree to maintain the shortest paths. Concretely, each count tree includes $2^{|q.key|} - 1$ nodes, and each node in the count tree corresponds to a non-empty subset of $q.key$. Most recently, Chen et al. [?] develop an index-based method to answer the why not top-k spatial keyword query, which enables to modify the query for including desired results. Wu et al. [?] study the authentication of moving top-k spatial keyword queries using the proposed MIR-tree, which is based on the IR-tree by embedding a series of digests in each node of the tree.

Combining with nearest neighbor queries. The nearest neighbor (NN) queries [? ?] are to retrieve the nearest object to one or a group of query points from the underlying database. By considering keywords, the spatial keyword NN queries return the closest object to the query point among those objects that contain the query keywords. Recently, several variations are explored. To eliminate the false hit problem of IR2-Tree [?], Tao et al. [?] propose the SI-index. SI-index is actually a compressed version of inverted list with the gap-keeping technique. Lu et al. [?] study the RSTkNN query, finding objects that take a specific query object as one of their k most spatial-textual similar objects. Jiang et al [?] study the k nearest keyword (k-NK) query in the context of large networks. They develop memory-based and disk-based exact methods using the label techniques.

Combining with route queries. The conventional route queries [?] search the shortest route that starts at location s, passes through as least one object from each category in C and ends at t. To be applicable to various applications, different approximate algorithms are proposed in [?]. Lee et al. [?] propose an initialization method for the robot path planning based on the directed acyclic graph. Yao et al. [?] investigate the multi-approximate-keyword routing (MARK) query. MARK asks for the shortest route that covers at least one matching object per keyword with the similarity greater than the given threshold. The problem of keyword-aware optimal route (KOR) search is studied in [?]. KOR retrieves the route that covers a set of given keywords, a specific budget constraint is satisfied, and an objective score of the route is optimal. Since the KOR query is NP-hard, thus three approximate algorithms are developed. In the subsequent work [?], they present the corresponding system. Cao et al. [?] are to find a length-constraint region (expressed by a connected subgraph of road networks) while best matching the query keywords. Approximate algorithms are developed to answer it due to the inherent complexity of this problem.

Besides, Bobed et al. [?] are to develop a system that can perform the keyword queries by considering the intended semantics of input keywords. It is clear that all these works are different from ours. Specifically, aforementioned works only return individual objects for satisfying users' needs. In contrast, our query aims to find a group of objects that satisfy users' needs collectively. Thus, their methods cannot be used to solve our problem. Subsequently, we review the collective keyword queries that are closer to our query compared with the above works.

2.2. Collective spatial Keyword Queries

The collective spatial keyword queries are studied to find a group of objects together covering the query keywords. The m-closed keywords (mCK) queries are studied in [? ?]. Given a query point constituted by a location and a set of keywords, mCK asks for a group of objects that together cover all keywords and have the minimum diameter. Specifically, they define the diameter to be the largest distance between any pair of objects in the group. Zhang et al. [?] answer the mCK query by traversing down the bR^*-tree in a depth-first manner. In addition, the priori-based search strategies are exploited to shrink the search space. Later, Zhang et al. [?] investigate the mCK query with the web data. The bottom-up strategy is developed to find a candidate answer, which prunes unnecessary accesses significantly. To further facilitate the query processing, Guo et al. [?] answer the mCK query approximately by searching the smallest diameter circle in which the final answer locates.

Cao et al. [?] extend the mCK query by considering other cost functions, and then study two variations of mCK. First, they aim to minimize the sum distances between the retrieved group of objects and query point. They develop a greedy method that finds the result by answering a sequence of partial queries progressively. In addition, an exact algorithm is proposed by exploiting the dynamic programming. Second, they investigate the cost function that considers both the inter-object distance and the distance between the object in the retrieved group and query point. A simple approximate algorithm is first proposed to address this problem. It retrieves the nearest object for each query keyword, and then merges them as the result set. Based on this method, another approximate method is developed by refining the retrieved group progressively. Finally, an exact algorithm is presented. Specifically, this exact algorithm first invokes the second approximate algorithm for deriving an upper bound, and then shrinks the search space with pruning strategies. Later, they extend this work by exploring other cost functions in [?]. Long et al. [?] study CoSKQ on two cost functions, namely the maximum sum cost and the diameter cost. With the distance owner-driven approach, they answer the query exactly and approximately.

Recently, Gao et al. [?] study CoSKQ on road networks based on the Connectivity-Clustered Access Method (CCAM) index. Skovsgaard et al. [?] propose to find top-k disjoint groups of objects while considering the group density, distance to the query, and relevance to the query keywords. Efficient algorithms are presented based on the Group Extended R-Tree, which is extended from the R-tree with each node including compassed histograms. Deng et al. [?] study the best keyword cover (BKC) query, which considers the keyword rating. To answer this query, keyword nearest neighbor expansion (keyword-NNE) algorithm returns the local best solution with the highest score as the answer. Wang et al. [?] propose to explore the spatial keyword query over streaming data with the AP-tree. All these works do not consider the level and have different query goals with ours.

3. Problem Statement

We first introduce basic notations to formalize the problem of the LCSK query. Let $O = \{o_1, ..., o_n\}$ be a spatial database. An object $o \in O$ consists of a spatial location $o.\ell$, a set of keywords $o.\omega$ capturing the textual description and a multidimensional level vector $o.\nu$ having $|o.\omega|$ elements. As mentioned in Section 1, the ith element of $o.\nu$, i.e., $o.\nu^i$, represents the level of corresponding keyword in $o.\omega$, i.e., $o.\omega^i$. For ease of presentation, hereon, we denote *keyword level* by *level* and consider it as a positive integer. In addition, we assume the level ranges from 1 to 5 (e.g., there are total 5 levels with respect to attractions). Actually, this

Notation	Explanation
q	an LCSK query of the form:$(\ell, \omega, W, \theta)$
O, o	the underlying database, an object in O
RO_q	relevant objects to q in O
G	the answer (i.e., a group of objects) of q
$\|S\|$	the cardinality of S
$cost(o)$	the cost of an object o
$cd(o, q)$	the cost distance between o and q
$cw(o, t)$	coverage weight of t by o
$cov(G, q)$	coverage weight of q by G
KHT	keyword hash table index structure
LIR-tree	index structure extended from the IR-tree
$cr(e, q), cr$	contribution ratio of the entry e to q, the abbreviation of contribution ratio
$dcr_q^r(e)$	dynamic contribution ratio of the entry e to q when $\|G\| = r$
dcr	the abbreviation of dynamic contribution ratio

Table 1. Summary of the notations

is not as restrictive as it seems, and the range can be extended to any scope. As with [?], we assume that each object o has a positive cost, namely $cost(o)$, which plays a critical role in decision support. Consider, for example, $cost(o)$ can capture the user ratings of products. Intuitively, the higher the rating, the more people will compete for such products, which incurs higher costs.

Definition 1. (Cost Distance) Given a query q and an object $o \in O$, the cost distance between o and q can be defined as:

$$cd(o, q) = cost(o) \cdot dist(o, q) \tag{1}$$

In Equation (??), $dist(o, q)$ represents the Euclidean distance between o and q. The cost distance is more realistic compared with the cost functions in [? ?], since the objects in the real world are always associated with the *cost* for highlighting some features.

Definition 2. (Coverage weight) Given a keyword t, a weight vector W and an object $o \in O$. We denote by $o.\nu_t$ the corresponding level of t in $o.\nu$. The coverage weight of t by o can be defined as:

$$cw(o, t) = W[o.\nu_t] \tag{2}$$

Different from CoSKQ in which t is either covered by an object o or not, in this work we consider the coverage weight. Note that if t is not contained by $o.\omega$, that is $t \notin o.\omega$ then we set $cw(o, t)$ to 0. We use $cov(o, q) = \sum_{t \in q.\omega} cw(o, t)$ and $cov(G, t) = \sum_{o \in G} cw(o, t)$ to denote the coverage weight of q by o and the coverage weight of t by a group G of objects, respectively. As depicted in Example 1, we know that $cov(o_1, q) = cw(o_1, t_1) + cw(o_1, t_2) = W[4] + W[5] = 0.25 + 0.3 = 0.55$. In addition, if $cw(o, t)$ is greater than the given threshold $q.\theta$ then we set it to $q.\theta$.

Definition 3. (Level-aware Collective Spatial Keyword Query). Given an LCSK query $q = (\ell, \omega, W, \theta)$, where ℓ denotes the location and ω is the query keywords. W is a user-specified weight vector and θ is a threshold. The answer of the LCSK query is a group G of objects that satisfy the following two conditions:

- $\forall t \in q.\omega$, it holds that $cov(G, t) \geq q.\theta$ (threshold constraint);
- $\arg\min_{G} \sum_{o \in G} cd(o, q)$.

entry	keyword	object list	first index	second index
e_1	t_1	o_1, o_3, o_7, o_{10}	5	0
e_2	t_2	o_3, o_9	6	1
e_3	t_3	o_1, o_7	6	2
e_4	t_4	o_2, o_4, o_5	6	1
e_5	t_5	o_1, o_3, o_6	1	0
e_6	t_6	o_2, o_7	2	0
e_7	t_7	o_5, o_6	5	1
e_8	t_8	o_5, o_{10}	0	0
e_9	t_9	o_8, o_{10}	4	0

Table 2. Example of KHT entries

Given a query q, we say that an object o is ***relevant*** to q if $o.\omega$ contains at least one keyword $t \in q.\omega$. In this paper hereafter, we denote by RO_q all the relevant objects in O to q, that is, $RO_q = \{o|o \in O \wedge (\exists t \in q.\omega \wedge t \in o.\omega)\}$. We then only need to consider RO_q for a specific query q. We call G a ***feasible solution*** to q if G satisfies the threshold constraint in Definition 3, i.e., $\forall t \in q.\omega$, $cov(G, t) \geq q.\theta$. In other words, the LCSK query returns the *feasible solution* with the minimum cost distance as the answer. Note that the answer to q may not be unique, since there may exist multiple feasible solutions with the same minimum cost distance. In this case, we return any one of them randomly. For ease of reference, Table **??** summarizes the notations used widely in this paper.

Theorem 1. *The LCSK query is NP-hard.*

Proof. *To prove the LCSK query is NP-hard, we reduce the WSC problem to it. Typically, an instance of the WSC problem of the form $< U, S, C >$, where $U = \{1, 2, 3, ..., n\}$ of n elements and a family of sets $S = \{S_1, S_2, S_3, ..., S_m\}$, where $S_i \subseteq U \wedge \cup S_i = U$. Each $S_i \subseteq U$ is associated with a positive cost $c_i \in C$ indicating the weight of S_i. The decision problem is to decide whether we can identify a subset $F \subseteq S$ such that $\cup_{S_i \in F} S_i = U$, and the sum of the cost of F, i.e., $\sum_{S_i \in F} c_i$, is minimized.*

Next, we reduce the WSC problem to an LCSK query q by two steps. We first construct the query q. We consider all elements in U as the query keywords $q.\omega$. Then, we set the weight vector $q.W$ as $\{1, 0, 0, 0, 0\}$ and the threshold $q.\theta$ to 1. Here, $q.\ell$ can be assigned with any location, which has no effect. Secondly, the spatial database O can be constructed as follows. We observe that each set S_i corresponds to an object o_i and the set of keywords $o_i.\omega$ is comprised of all elements in S_i. We then assign c_i as the cost distance $cd(o_i, q)$ and set each level of $o_i.\nu$ to 1. Then, it is clear that there is a solution to the WSC problem if and only if there is an answer to the query q. Hence, we complete the proof.

4. Exact Algorithm

The discussion in Section 3 suggests that we only need to consider RO_q for a specific query q. This motivates us to develop an exact algorithm in the context where $|RO_q|$ is limited. In this section, we study the exact algorithm MergeList. Concretely, we first introduce the flat index structure keyword hash table (KHT) in Section 4.1, and then elaborate MergeList in Section 4.2.

4.1. The KHT Index Structure

For a user-specified query q, only the objects in RO_q need to be explored. We thus propose the keyword inverted index structure, which facilitates the access to the objects in RO_q. Specifically, the keyword inverted index structure uses the hash table to organize the objects in O based on the *perfect hashing technique*. For brevity, this index structure is abbreviated to KHT hereafter.

As shown in Table **??**, each entry of KHT of the form $< eid, t, olist, fi, si >$, where eid is the identifier of an entry and t denotes a keyword followed by a list of objects $olist$. Here, $olist$ maintains all relevant objects

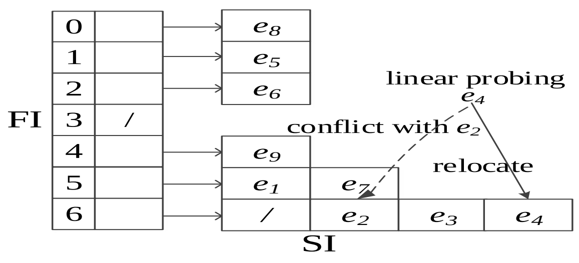


Fig. 2. An overview of KHT structure

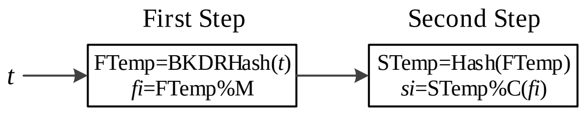


Fig. 3. Construction of a two-level index

in O to t, i.e., $olist = \{o | o \in O \wedge t \in o.\omega\}$. The fi and si indicate the first and second index subscripts of an entry, respectively. That is, we identify an entry of KHT by a two-level index. As an example, Fig. **??** presents an instance of KHT index over the entries in Table **??**. For reason of space, we denote by *eid* the corresponding entry in Fig. **??**.

To manage the trade-off between exploration time and space consumption, we use the perfect hashing technique [**?**] to build the two-level index. In particular, each entry e is projected to a two-level index (fi, si), and thus can be accessed in $O(1)$ time. We proceed to show how to build a two-level index for the keyword t.

As shown in Fig. **??**, we identify a two-level index in two steps. First, we project t to a unique integer $FTemp$ using the string hash function BKDRHash. Note that, other string hash functions can also be exploited. Next, we derive fi by performing a modulus on $FTemp$ over M, where M is the predefined cardinality of the first level index. That is, the range of the first index varies from 0 to $M - 1$. Second, we take $FTemp$ as the parameter and generate $STemp$ using an integer hash function. Similarly, we perform a modulus on $STemp$ over $C(fi)$ to get si. Here, $C(fi)$ indicates the capacity of the first level index fi, which is dynamically determined based on the keyword distribution of the dataset. By these two steps, each entry is associated with a two-level index, denoted by (fi, si). It is noteworthy that a delicate situation arises when multiple entries share the same two-level index (fi, si), and the possibility is less than 0.5 as pointed out by [**?**]. We use the *linear probing* technique to tackle this problem. For instance, there is a conflict when we attempt to insert e_4 into KHT since e_2 has already been in the location $(6, 1)$. As the location $(6, 2)$ is occupied as well, with the linear probing technique, e_4 is inserted into $(6, 3)$ ultimately, as shown by the solid arrow.

From the above discussion, we know that KHT organizes the objects in a flat structure, which facilitates the access to relevant objects using the two-level index (fi, si). In particular, given a query q, we can retrieve RO_q in $O(|q.\omega|)$ time and avoid unnecessary accesses significantly.

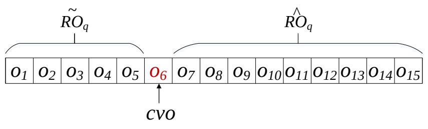
4.2. Query Processing of MergeList

We first consider a naive exact algorithm that enumerates all subsets of RO_q, and then delivers the one that satisfies the threshold constraint and has the minimum cost distance as the answer.

Fig. **??** shows how to enumerate all subsets iteratively. Initially, the *candidate set* $\tilde{C}$ includes only one element, i.e., the empty set $\{\}$. In each of the subsequent iterations, we generate a new set (e.g., $\{o_1\}$) for each existing set (e.g., $\{\}$) in $\tilde{C}$ by inserting the current visited object (e.g., o_1) into it. Clearly, this straightforward method yields an exponential time complexity in terms of $|RO_q|$. The bottleneck lies in that all subsets of RO_q have to be enumerated and checked for finding the answer. In practice, however, a large

Step	Action	$\tilde{C}$
1	initialization	{}
2	visit o_1	{},{o_1}
3	visit o_2	{},{o_1},{o_2},{o_1,o_2}
.	.	.
.	.	.
.	.	.

Fig. 4. Illustration of the naive approach



$\tilde{RO}_q$
$\hat{RO}_q$
o_1 o_2 o_3 o_4 o_5 o_6 o_7 o_8 o_9 o_{10} o_{11} o_{12} o_{13} o_{14} o_{15}
cvo

Fig. 5. Illustration of the RO_q

number of the subsets are unqualified considering the cost distance. That is, they can be pruned by the best answer found so far if they have an equal or greater cost distance. Subsequently, we give a high-level overview on the enhanced exact algorithm MergeList.

MergeList answers the LCSK query by first constructing RO_q where the objects are placed in ascending order of their cost distances, then expanding the candidate set $\tilde{C}$ iteratively with the objects in RO_q until the termination condition (to be presented in Lemma 2) is achieved. Here, we keep the best answer (a group of objects in RO_q) found so far and its cost distance in COS and $minCost$, and finally delivering COS as the answer. MergeList enhances the naive approach from the following two aspects. First, MergeList organizes the objects in RO_q in ascending order of their cost distances. This enables two effective pruning strategies, namely Lemmas 1 and 2, to significantly shrink the candidate space. Second, $\tilde{C}$ only maintains the promising answers in MergeList, whereas all the subsets of RO_q are maintained in the naive approach. We denote a *promising answer* $\xi \in \tilde{C}$ by a group of objects in RO_q satisfying two conditions: (1) the threshold constraint in Definition 3 is not satisfied, that is, $\exists t \in q.\omega \wedge cov(\xi, t) < q.\theta$. As for ξ that satisfies the threshold constraint, we know that it is a feasible solution. Hence, we can assign it as COS or prune it by COS, and (2) the cost distance of ξ is less than $minCost$, i.e., $\sum_{o\in\xi} cd(o,q) < minCost$. As for ξ that has an equal or greater cost distance, we can eliminate them by Lemma 1. Thus, they do not need to be reserved in $\tilde{C}$.

We proceed to introduce some notations used by the following lemmas. As shown in Fig. **??**, the current visited object *cvo* splits the ordered RO_q into three components, namely $\tilde{RO}_q$, *cvo* and $\hat{RO}_q$. We denote by $\tilde{RO}_q$ the objects in RO_q that have already been visited (before *cvo*). Similarly, $\hat{RO}_q$ represents the objects that have not been visited (after *cvo*). In particular, given two objects o, $o' \in RO_q$, we denote by $o \prec o'$ ($o \preceq o'$) that o is after (not before) o' in RO_q.

We claim that a promising answer ξ *precedes cvo* if all its elements are in $\tilde{RO}_q$, denoted by $\xi|_{cvo}$. Put differently, $\forall o \in \xi$, it holds that $cov \prec o$. Consider, for example, the promising answer $\{o_1, o_5\}$ precedes o_6 in Fig. **??** because $o_6 \prec o_1$ as well as $o_6 \prec o_5$. Given a promising answer ξ (e.g., $\{o_1, o_5\}$) that precedes *cvo*, we say that the set ξ' (e.g., $\{o_1, o_5, o_8\}$) is a successor of ξ with respect to *cvo* (e.g., o_6), if $\xi \subset \xi'$ and $\forall o \in \xi' \wedge o \notin \xi$, it holds that $o \preceq cvo$. From Fig. **??**, we know that $\{o_1, o_5, o_8\}$ is a successor of $\{o_1, o_5\}$ with respect to o_6. Specifically, each promising answer will later spawn an exponential number of successors. We denote by Γ_ξ^{cvo} all the successors of ξ with respect to *cvo* hereafter.

Lemma 1. *Given a promising answer $\xi \in \tilde{C}$ and the current visited object cvo $\in RO_q$. If the sum of the cost distance of ξ and cvo is greater than minCost, that is, $\sum_{o\in\xi} cd(o,q) + cd(cvo,q) > minCost$, then no successor $\xi' \in \Gamma_\xi^{cvo}$ can be the answer of q.*

Proof. *We prove this lemma by contradiction. Let $\xi' \in \Gamma_\xi^{cvo}$ be the answer returned by MergeList. Thus, we have $\sum_{o\in\xi'} cd(o,q) \leq minCost$. By the property of successor, we know that there exists at least one object o', and it holds that $o' \in \xi' \wedge o' \notin \xi$. Also, we know that $o' \preceq cov$, and thus $cd(cov,q) \leq cd(o',q)$. Combining the above two inequalities together with the given condition $\sum_{o\in\xi} cd(o,q) + cd(cvo,q) > minCost$, we have $\sum_{o\in\xi'} cd(o,q) = \sum_{o\in\xi} cd(o,q) + \sum_{o'\in\xi'\wedge o'\notin\xi} cd(o',q) > minCost$. This contradicts the assumption that ξ' is the final answer. Thus, no successor $\xi' \in \Gamma_\xi^{cvo}$ can be the answer. We complete the proof.*

Lemma 1 enables to significantly shrink the candidate space, i.e., the size of $\tilde{C}$. As suggested in Lemma 1, if the sum of the cost distance of ξ and cvo is greater than $minCost$, then we can discard ξ and all its successors Γ_{ξ}^{cvo} from consideration. This is because the solution quality of COS is always better than or equal to that of any set in $\xi \cup \Gamma_{\xi}^{cvo}$. Subsequently, we reveal an appealing property that allows to terminate the query processing.

Algorithm 1: *MergeList*

```
Input  : The KHT and the query $q$.
Output: A group $G$ of objects.
1  $COS \longleftarrow \{\}$;
2  add the empty set $\{\}$ into $\tilde{C}$;
3  $minCost \longleftarrow \infty$;
4  $RO_q \longleftarrow$ retrieve the relevant objects to $q$ from KHT;
5  place the objects in $RO_q$ in ascending order of their cost distances;
6  for each object $cvo \in RO_q$ do
7      if $cd(cvo, q) > minCost$ then
8          break;
9      for each $\xi \in \tilde{C}$ do
10         if $\sum_{o \in \xi} cd(o, q) + cd(cvo, q) > minCost$ then
11             delete $\xi$ from $\tilde{C}$;
12             continue;
13         $tempSet \longleftarrow \xi \cup \{cvo\}$;
14         if $tempSet$ is a feasible solution then
15             $COS \longleftarrow tempSet$;
16             $minCost \longleftarrow$ the cost distance of $tempSet$;
17             delete $\xi$ from $\tilde{C}$;
18         else
19             add $tempSet$ into $\tilde{C}$;
20     if $\tilde{C} == \emptyset$ then
21         break;
22 $G \longleftarrow COS$;
23 return $G$ as the answer;
```

Lemma 2. *Given the ordered RO_q, in which the objects are placed in ascending order of their cost distances. If $\forall \xi \in \tilde{C}$, the sum of the cost distance of ξ and cvo is greater than minCost, i.e., $\sum_{o \in \xi} cd(o, q) + cd(cvo, q) > minCost$, then COS is the answer.*

PROOF. *As noted earlier, $\tilde{C}$ maintains all the promising answers found so far. Let ξ be any promising answer in $\tilde{C}$. By the property of the promising answer, we know that ξ is not a feasible solution, and additional objects that are not before cvo, i.e., $o' \preceq cvo$, are required to satisfy the threshold constraint. On the other hand, we know that $\forall o' \preceq cvo$, it holds that $cd(o', q) \geq cd(cvo, q)$. Based on the given condition $\sum_{o \in \xi} cd(o, q) + cd(cvo, q) > minCost$, then we derive that $\sum_{o \in \xi} cd(o, q) + cd(o', q) > minCost$ as well. Combining the above inequality together with Lemma 1, we derive that there does not exist unseen feasible solutions that achieve a smaller cost distance than minCost. Thus, COS is the final answer and we complete the proof.*

We elaborate the details of MergeList in Algorithm 1. We start by constructing the ordered RO_q (lines 4-5), in which the objects are placed in ascending order of their cost distances. We then access the objects in RO_q iteratively as follows. Consider an iteration. We first examine whether the cost distance of *cvo* is greater than *minCost*. If so, for $\forall \xi \in \tilde{C}$, we know that the sum of the cost distance of ξ and $cd(cvo, q)$ is greater than *minCost* as well. Based on Lemma 2, we know that *COS* is the answer, thus we terminate the procedure (lines 7-8). Otherwise, each $\xi \in \tilde{C}$ is processed as follows. Firstly, we verify whether Lemma 1 is satisfied. If yes, we delete ξ from $\tilde{C}$ (lines 10-12). If no, we generate a new set by adding *cvo* into ξ (line 13). In particular, if *tempSet* is a feasible solution, we then update *COS* and *minCost* accordingly by *tempSet* (lines 14-17). Otherwise, we put *tempSet* into $\tilde{C}$ for further exploration. If $\tilde{C}$ is empty then we stop the procedure and deliver G as the answer (lines 20-23).

Object id	o_1	o_2	o_3	o_4	o_5
Cost distance to q	2	2.5	4	5	7
Coverage weight	0.1, 0.0	0.1, 0.3	0.3, 0.1	0, 0.3	0.1, 0.3

(a) Example dataset

Step	Action	Candidate Set	*minCost*	*COS*
1	initialization	{}	∞	{}
2	visit o_1	{},$\{o_1\}$	∞	{}
3	visit o_2	{},$\{o_1\}$,$\{o_2\}$,$\{o_1, o_2\}$	∞	{}
4	visit o_3	{},$\{o_1\}$,$\{o_2\}$,$\{o_1, o_2\}$,$\{o_3\}$,$\{o_1, o_3\}$	6.5	$\{o_2, o_3\}$
5	visit o_4	{},$\{o_4\}$	6.5	$\{o_2, o_3\}$
6	visit o_5	{},$\{o_4\}$	6.5	$\{o_2, o_3\}$

(b) Example query processing steps

Fig. 6. Illustration of the MergeList algorithm

Example 2: Consider a query q associated with two query keywords, e.g., $q.\omega = \{t_1, t_2\}$. Fig. **??**(a) depicts the objects in RO_q, the cost distance to q and the coverage weight of these two keywords by objects in RO_q. In Fig. **??**(b), we present how to answer the query by MergeList. There are six steps.

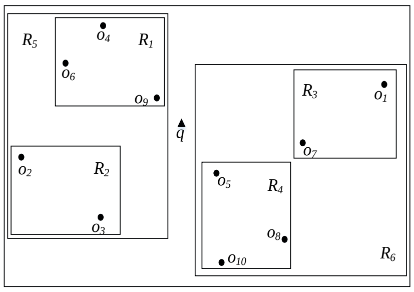
- Step 1 (initialization): We initialize the Candidate Set $\tilde{C}$, *minCost* and *COS* at this step.
- Step 2 (visit o_1): At this step, we generate a new set for each promising answer in $\tilde{C}$ by adding o_1 into it. For instance, we add o_1 into the promising answer {} and obtain a new promising answer $\{o_1\}$.
- Step 3 (Visit o_2): Similarly, at this step, we add o_2 into existing promising answers in $\tilde{C}$ as depicted in Fig. **??**(b).
- Step 4 (visit o_3): At this step, we derive a feasible solution, i.e., $\{o_2, o_3\}$. We assign *COS* as $\{o_2, o_3\}$, and then prune the unqualified elements, e.g., $\{o_1, o_2, o_3\}$ by Lemma 1. Note that, we delete $\{o_2, o_3\}$ from $\tilde{C}$ as well. This is because the successors of $\{o_2, o_3\}$ have a greater cost distance than $\{o_2, o_3\}$, and they cannot be the answer. Thus, we can stop the extension from $\{o_2, o_3\}$ by deleting it from $\tilde{C}$.
- Step 5 (visit o_4): At this step, we prune unqualified elements of $\tilde{C}$ by Lemma 1. For instance, as the sum of the cost distance of $\{o_1, o_2\}$ and $\{o_4\}$ is greater than *minCost*, then we delete $\{o_1, o_2\}$ from $\tilde{C}$.
- Step 6 (visit o_5): Because the cost distance of o_5 is greater than *minCost*, thus Lemma 2 is applied and we terminate the procedure. We deliver *COS*, that is, $\{o_2, o_3\}$, as the answer.

Theorem 2. *(Correctness of MergeList) MergeList always returns the correct answer.*

PROOF. *Let the cardinality of RO_q be n, thus, there are up to $2^n - 1$ non-empty elements in $\tilde{C}$. For any $\xi \in \tilde{C}$, it is either used to update COS or pruned by COS. Specifically, if the sum of the cost distance of ξ and cvo is less than minCost, we update COS by $\xi \cup \{cvo\}$. Otherwise, we delete ξ from $\tilde{C}$ based on Lemma 1. Hence, it suffices to show that MergeList never prunes any feasible solutions with a smaller cost distance than minCost (false negatives), and never maintains feasible solutions with a greater cost distance than minCost (false positives). Combining the above two observations, we safely drive that MergeList always returns the correct answer.*

5. Proposed Approximate Algorithm

Though equipped with some pruning strategies, however, MergeList needs to produce a large number of possible subsets for answering a query, which is prohibitively expensive when the dataset is large. In this



Object descriptions			
object	keywords	level vector	cost
o_1	t_1, t_3	3, 1	0.4
o_2	t_1, t_6	1, 1	0.2
o_3	t_2, t_5	5, 2	0.7
o_4	t_3, t_7	2, 3	0.5
o_5	t_4, t_7	2, 4	0.6
o_6	t_3, t_5	5, 3	0.8
o_7	t_2, t_6	1, 2	0.3
o_8	t_5, t_6	1, 5	0.6
o_9	t_1, t_7	3, 2	0.5
o_{10}	t_6, t_7	4, 3	0.7

Fig. 7. Example spatial database

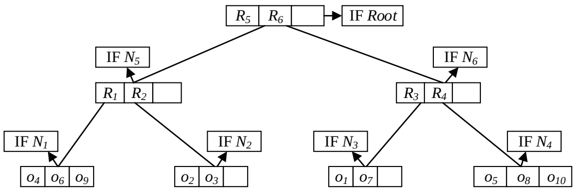


Fig. 8. Example LIR-tree

section, we propose an approximate algorithm called MaxMargin to facilitating the query processing under various settings. In the following, we introduce the hierarchical index structure LIR-tree in Section 5.1 and elaborate pruning strategies in Section 5.2. Finally, we delve into the details of MaxMargin in Section 5.3 with theoretical analysis.

5.1. The LIR-tree Index Structure

We proceed to briefly review the IR-tree [**?**], which lays the foundation of the LIR-tree index. The IR-tree is essentially an R-tree [**?**] with each node augmented with an inverted file [**?**]. Each node n of the IR-tree contains multiple entries. Each entry e of the form $(id, mbr, text)$, where id is an identifier of a child node (an object) of the non-leaf (leaf) node, mbr indicates the minimum bounding rectangle (MBR) enclosing all rectangles of the entries in n, and $text$ captures the text description of all entries in n. In addition, each node is associated with an inverted file constituted by two components, namely the vocabulary table and the posting list, as follows:

- Vocabulary table: Vocabulary table includes all distinct keywords contained by the textual descriptions of objects in the underlying database.
- Posting list: Each distinct keyword t has a posting list, which maintains all the ids of the entries (i.e., the objects or the child nodes) with t in their descriptions.

It is clear that IR-tree organizes spatial objects in the hierarchical structure, which enables to refine the search space in higher hierarchical level. Consider, for example, we can first check whether n satisfies the query constraints, if so we expand it by adding all its child nodes into the priority queue for further exploration. Otherwise, we skip the entire subtree rooted at n. In the sequel, we show how to build the LIR-tree based on the IR-tree.

In general, the LIR-tree augments the IR-tree with the information of levels and costs. Specifically, for each posting list, we embed the level into each entry and maintain the lowest cost among all the entries in the posting list. Each posting list in the leaf node of the form $< lowCost; (oid, level) >$, where $lowCost$ denotes the lower bound of the cost of all objects in the posting list, and $(oid, level)$ refers to the identifier of an object and the corresponding level of a keyword. We consider IF N_4 in Fig. **??** as an example. We observe that the posting list of t_4 contains two objects o_5 and o_{10}. As shown in Fig. **??**, the cost of o_5 is 0.6,

IF N_1	IF N_2	IF N_3	IF N_4	IF N_5	IF N_6	IF $Root$
t_1:0.5;(o_9,3)	t_1:0.2;(o_2,1)	t_1:0.4;(o_1,3)	t_4:0.6;(o_5,2),(o_{10},4)	t_1:0.2;(N_1,0.5),(N_2,0.2)	t_1:0.4;(N_3,0.4)	t_1:0.2;(N_5,0.2),(N_6,0.4)
t_3:0.5;(o_4,2),(o_6,5)	t_2:0.7;(o_3,5)	t_2:0.3;(o_7,1)	t_5:0.6;(o_8,1)	t_2:0.7;(N_2,0.7)	t_2:0.3;(N_3,0.3)	t_2:0.3;(N_5,0.7),(N_6,0.3)
t_5:0.8;(o_6,3)	t_5:0.7;(o_3,2)	t_3:0.4;(o_1,1)	t_6:0.6;(o_8,5)	t_3:0.5;(N_1,0.5)	t_3:0.4;(N_3,0.4)	t_3:0.4;(N_5,0.5),(N_6,0.4)
t_7:0.5;(o_4,3),(o_9,2)	t_6:0.2;(o_2,1)	t_6:0.3;(o_7,2)	t_7:0.6;(o_5,4),(o_{10},4)	t_5:0.7;(N_1,0.8),(N_2,0.7)	t_4:0.6;(N_4,0.6)	t_4:0.6;(N_6,0.6)
				t_6:0.2;(N_2,0.2)	t_5:0.6;(N_4,0.6)	t_5:0.6;(N_5,0.7),(N_6,0.6)
				t_7:0.5;(N_1,0.5)	t_6:0.3;(N_3,0.3),(N_4,0.6)	t_6:0.2;(N_5,0.2),(N_6,0.3)
					t_7:0.6;(N_4,0.6)	t_7:0.5;(N_5,0.5),(N_6,0.6)

Fig. 9. Content of augmented inverted files

which is less than the cost of o_{10}, i.e., 0.7. Thus, we set $lowCost$ to 0.6. Also, we know that o_5 is associated with two keywords t_4 and t_7, and the level of t_4 is 2. Thus, we store the pair $(o_5, 2)$ in the posting list of t_4.

Each posting list in the non-leaf node of the form $< lowerCost; (cid, lowCost) >$, where $lowerCost$ denotes the lower bound of the $lowCost$ of all child nodes in the posting list, and $(cid, lowCost)$ refers to an identifier of a child node and the $lowCost$ of corresponding posting list in cid. We take IF N_6 in Fig. **??** as an example. We observe that the posting list of t_6 contains two entries, i.e., $(N_3, 0.3)$ and $(N_4, 0.6)$. That is to say that the $lowCost$ of N_3 is 0.3 with respect to t_6. We can derive the value from the posting list of t_6 in IF N_3. Then, we set the $lowerCost$ to 0.3 because it is the lower bound of the $lowCost$ of N_3 and N_4, i.e., it holds that $0.3 < 0.6$. For simplicity, we denote by $lowerCost$ the $lowCost$ (in leaf node) and $lowerCost$ (in non-leaf node) hereafter.

5.2. Pruning Strategies

To address the WSC problem, Chvatal [**?**] proposes a greedy strategy, which iteratively appends the current optimal subset into the result set and then updates the remaining subsets accordingly. This strategy is also adopted by [**? ?**] for answering the spatial keyword query. The greedy algorithms in [**? ?**] find the current optimal entry by traversing the IR-tree in the best-first fashion. Initially, the root node of IR-tree is pushed into the priority queue Q. In each of the subsequent iterations, the top entry e is popped and examined as follows. If e is an object then we add it into the result set G and update all the remaining entries in Q accordingly. Otherwise, all the child nodes of e are pushed into Q for further exploration.

Two major drawbacks degrade the performance of the above algorithms: (1) the pruning power of the above algorithms is insufficient, and (2) they have to update all the remaining entries in each iteration, which is computationally expensive. We develop the *branch and bound strategy* and the *triggered update strategy* to attack the aforementioned drawbacks accordingly. Subsequently, we take a look at the basic concepts that lay the foundation of these strategies.

Definition 4. (Minimum Cost Distance) Given a query q and a node n of the LIR-tree, we define the minimum cost distance between n and q as:

$$mcd_q(n) = minDist(q, n) * lowerCost(q, n) \tag{3}$$

In Definition 4, we denote by $minDist(q, n)$ the possible minimum distance between q and the MBR of n, and $lowerCost(q, n)$ denotes the lowest cost among all relevant objects in n. Recall that, we have maintained $lowerCost$ in the posting list, and thus $lowerCost(q, n)$ can be retrieved from the augmented inverted file (see Fig. **??**) quickly. For simplicity, *we say that an object o is in n if o is enclosed by the MBR of n hereafter*, whenever there is no ambiguity. Based on the above discussion, we know that $mcd_q(n)$ is the lower bound of the cost distance of all objects in n.

Lemma 3. *Let FS be a feasible solution to a query q, $\hat{FS}$ be the corresponding cost distance, and n be a node of the LIR-tree. If $mcd_q(n) > \hat{FS}$, then no object in n can be in the answer.*

PROOF. *Recall that, $mcd_q(n)$ offers a lower bound of the cost distance of all objects in n. That is, for any object o in n, it holds that $cd(o, q) \geq mcd_q(n)$. Combining the given condition $mcd_q(n) > \hat{FS}$, thus we* *derive $cd(o, q) > \hat{FS}$ as well. As FS is a feasible solution to q, and it achieves better performance than any other feasible solutions that contain the objects in n in terms of the cost distance. As a result, no object in n can be in the answer and thus can be discarded from consideration. Hence, we complete the proof.*

Branch and Bound Strategy. As suggested in Lemma 3, we can take $\hat{FS}$ as the upper bound and refine the search space with it. In particular, we examine each node n as follows. If $mcd_q(n) > \hat{FS}$, we then skip the entire subtree rooted at n and all objects in n are pruned. Otherwise, we add all the child nodes of n into the priority queue Q for further exploration. Next, we show how to build the initial feasible solution.

We build the initial feasible solution with the relevant objects that are close to q. Here, the best-first search strategy of the NN query is adopted. We first start from the root node of LIR-tree, and then find the unseen *nearest leaf node* iteratively in the best-first fashion. We refer to the *nearest leaf node* as a leaf node that has the minimum distance, i.e., $minDist(q, n)$, to q and contains at least one query keyword. When we find a nearest leaf node n, then all relevant objects in n are organized in ascending order of their cost distances to q. We add these objects into FS progressively until the threshold constraint is satisfied. That is, FS is a feasible solution. Note that, we continue to find the next unseen nearest leaf node if the threshold constraint is not satisfied. We offer a tighter $\hat{FS}$ by updating the feasible solution when an object is added into the result set G. Clearly, the *branch and bound strategy* facilitates the query processing by reducing the cardinality of the priority queue Q.

Before describing in detail how to use the triggered update strategy to further improve the query performance, we explain some concepts as follows.

Definition 5. (Contribution ratio) Given a query q and an entry e (i.e., an object or a node) of the LIR-tree, we define the contribution ratio cr of e to q as follows:

$$cr(e, q) = \begin{cases} \frac{|q.\omega| * q.\theta}{mcd_q(e)} & \text{if e is a node,} \\ \frac{cov(e,q)}{cd(e,q)} & \text{if e is an object.} \end{cases}$$

In Definition 5, we denote by $|q.\omega|$ the number of query keywords. By considering both the coverage weight and the cost distance, $cr(e, q)$ can better capture the contribution of e to q than that of $cov(e, q)$. Note that the contribution ratio of an unseen entry e in Q decreases as the objects are added into G. To capture this change dynamically, we define the *dynamic contribution ratio* as follows.

Definition 6. (Dynamic contribution ratio) Given a query q and an entry e (i.e., an object or a node) of the LIR-tree, we define the dynamic contribution ratio dcr of e when there are r objects in the result set G, i.e., $|G| = r$, as follows:

$$dcr_q^r(e) = \begin{cases} \frac{remain_q^r * q.\theta}{mcd_q(e)} & \text{if e is a node,} \\ \frac{cov^r(e,q)}{cd(e,q)} & \text{if e is an object.} \end{cases}$$

In Definition 6, we denote by $remain_q^r$ the number of query keywords whose coverage weight have not reached the threshold $q.\theta$. The $cov^r(e, q)$ represents the remaining coverage weight of q by the object e when r objects are included in G. Note that, both $remain_q^r$ and $cov^r(e, q)$ keep decreasing when we add an object into G.

Lemma 4. *Given a query q, a node n of the LIR-tree. For any object o in n, and any r ($r \geq 0$), it holds that $dcr_q^r(o) \leq dcr_q^r(n)$.*

Proof. *For any object o in n, we know that $cov^r(o, q) \leq remain_q^r * q.\theta$. Then, according to the definition of $mcd_q(n)$, we know that $mcd_q(n)$ is the lower bound of the cost distance of all objects in n, that is $cd(o, q) \geq mcd_q(n)$. With these two inequalities we derive the inequality $dcr_q^r(o) \leq dcr_q^r(n)$. Hence, the proof finishes.*

Lemma 4 provides an upper bound for the dynamic contribution ratio of all the objects in n. Subsequently, we reveal an appealing property of the dynamic contribution ratio in Lemma 5. Based on these two lemmas, we propose the triggered update strategy that reduces the updating cost of the priority queue Q.

Algorithm 2: *MaxMargin*

```
Input  : The query $q$.
Output: A group $G$ of objects.
$r \longleftarrow 0$; $G \longleftarrow \emptyset$;
for $i \longleftarrow 1$ to $|q.\omega|$ do
    $RV[i] \longleftarrow q.\theta$;
construct the feasible solution $FS$ with the objects near to $q$;
$upBound \longleftarrow \sum_{o \in FS} cd(o,q)$;
$Q.enqueue(LIR.root)$;
while $(!Q.empty())$ do
    $e \longleftarrow Q.dequeue()$;
    if $e$ is a node then
        recompute the $dcr_q^r(e)$;
        if $dcr_q^r(e)$ is greater than the dcr of the top entry in $Q$ then
            for each entry $e'$ in node $e$ do
                if $e'$ contains the query keywords and $mcd_q(e') < upBound$ then
                    compute the $dcr_q^r(e')$;
                    $Q.enqueue(e')$;
        else
            $Q.enqueue(e)$;
    else
        if $e.cv[i] \leq RV[i]$ for any $i$ then
            $G \longleftarrow G \cup \{e\}$;
            for $j \longleftarrow 1$ to $|q.\omega|$ do
                if $RV[j] \geq e.cv[j]$ then
                    $RV[j] \longleftarrow RV[j] - e.cv[j]$;
                else
                    $RV[j] \longleftarrow 0$;
            if $RV[j] == 0$ for any $j$ then
                break;
            $r$++;
            $FS \longleftarrow refineFS(FS, G)$;
            $upBound \longleftarrow \sum_{o \in FS} cd(o,q)$;
        else
            for $j \longleftarrow 1$ to $|q.\omega|$ do
                if $RV[j] < e.cv[j]$ then
                    $e.cv[j] \longleftarrow RV[j]$;
            recompute the $dcr_q^r(e)$;
            $Q.enqueue(e)$;
return $G$ as the answer;
```

Lemma 5. *Given a query q, an entry e (i.e., an object or a node) of the LIR-tree and two integers m, n ($m \leq n$), it holds that $dcr_q^n(e) \leq dcr_q^m(e)$.*

Proof. *As suggested by the formula of dynamic contribution ratio, the denominator (e.g., cd(e,q)) is a constant. In contrast, the numerator (e.g., $cov^r(e,q)$) may decrease when an object is added into G. As a* *result, we have $cov^n(e,q) \leq cov^m(e,q)$ and $remain_q^n \leq remain_q^m$, which leads to the decrease of dynamic contribution ratio. When $m \leq n$, we can safely draw the conclusion that $dcr_q^n(e)$ is not greater than $dcr_q^m(e)$. Hence, we complete the proof.*

Triggered Update Strategy. Previous approaches in [**?** **?**] update all the remaining entries in Q when an object is added into G, which is indeed time consuming and unnecessary. In contrast, the triggered update strategy does not update the dynamic contribution ratio of an entry e until it is popped from the priority queue Q. Lemma 5 suggests that if e is still the current optimal entry among all the remaining entries in Q, we can handle it as follows. If e is an object, we then add it into G. Otherwise, we expand it by pushing all its child nodes into Q for further exploration. Specifically, all the remaining entries in Q do not need to be updated, which significantly reduces the updating overhead. Here, the *triggered update strategy* improves the query performance by reducing the updating overhead.

5.3. Query Processing of MaxMargin

Based on the above two pruning strategies, we develop the approximate algorithm MaxMargin. MaxMargin answers the query by adding the current optimal object into the result set G progressively until the threshold constraint is satisfied. Next, we present the details of MaxMargin in Algorithm 2.

MaxMargin keeps the relevant entries (i.e., the objects and nodes) in Q in descending order of the dynamic contribution ratio dcr, and uses RV to record the difference between $q.\theta$ and the coverage weight of G for each query keyword dynamically. Initially, each dimension of RV, e.g., $RV[i]$, is set to $q.\theta$ (lines 2-3). Then, MaxMargin builds the initial feasible solution FS as we have described in Section 5.2. We set *upBound* as the cost distance of FS, and consider it as the upper bound. Next, we perform an iterative procedure as follows. We first pop the top entry e and find out whether e is a node. If yes, we continue as follows. Firstly, we need to recompute the dynamic contribution ratio of e, i.e., $dcr_q^r(e)$. Secondly, if $dcr_q^r(e)$ is greater than the dynamic contribution ratio of the top entry in Q, that is to say that e is the current optimal entry. Then, we push all unfiltered entries of e into Q (lines 11-15). Otherwise, e is pushed into Q (line 17). If no, that is, e is an object, there are two different cases. Case 1: If $\forall i$, it holds that $e.cv[i] \leq RV[i]$. Here, we denote by $e.cv[i]$ the remaining coverage weight of the ith keyword in $q.\omega$ by e. We know that the dynamic contribution ratio of e has not decreased, and thus e does not need to be updated. We thus add e into G and update RV accordingly (lines 20-25). Then, if the threshold constraint is achieved (line 26), we terminate the procedure immediately. Otherwise, we invoke Algorithm 3 to update FS and *upBound* (lines 29-30). Case 2: If $\exists i$, and it holds that $e.cv[i] > RV[i]$, we update RV and then push the updated e into Q (lines 32-36).

Algorithm 3 shows how to update FS by G. In this procedure, we consider a simple strategy. First, we combine FS with G and organize all objects in descending order of their cost distances (lines 1-2). Then, we remove the current object o from FS and obtain a temporary set, i.e., *Temp*. We check whether $Temp$ is a feasible solution. If so, we then set FS as $Temp$ (lines 4-6). The algorithm repeats the above procedure until all objects in FS have been explored. Finally, we return the updated FS as a new feasible solution.

Algorithm 3: *refineFS*

```
   Input  : $FS$ and $G$.
   Output: A new feasible solution $FS$.
 1 $FS \longleftarrow FS \cup G$;
 2 organize the objects in $FS$ in descending order of their cost distances;
 3 for each object $o \in FS$ do
 4     $Temp \longleftarrow FS - \{o\}$;
 5     if $Temp$ is a feasible solution then
 6         $FS \longleftarrow Temp$;
 7 return $FS$ as a new feasible solution;
```

Theorem 3. *The approximation ratio of MaxMargin is not greater than $\frac{H(\lfloor cov+1 \rfloor)}{q.\theta}$, where cov is the largest $cov(o_j, q)$ for $\forall o_j \in RO_q$, $cov+1$ is rounded down by $\lfloor cov+1 \rfloor$, and $H(k) = \sum_{i=1}^{k} \frac{1}{i}$ is the kth harmonic number.*

PROOF. *Inspired by [**?**], we present the proof of this theorem as follows. We denote by m, n the number of elements in $q.\omega$ and RO_q, respectively. We define the $m \times n$ matrix $P = (p_{ij})$ as follows:*

$$p_{ij} = \begin{cases} cw(o_j, q.\omega^i) & \text{if } q.\omega^i \in o_j.\omega, \\ 0 & \text{otherwise.} \end{cases}$$

Based on the definition of P, we know that n columns of P correspond to n coverage weight vectors. MaxMargin is to find a group G of objects, and we use an incidence vector $x = (x_j)$ to represent feasible solutions. As a result, the incidence vector x of any feasible solution satisfies:

$$\begin{aligned} \sum_{j=1}^{n} p_{ij}x_j \geq q.\theta \quad & \text{for all } i, \\ x_j \in \{0,1\} \quad & \text{for all } j. \end{aligned}$$

The formula above suggests that the group of objects x implied can satisfy the threshold constraint. For ease of presentation, we denote the cost distance of o_j by c_j hereafter. Also, we denote by cov_j^r the remaining coverage weight to q by o_j when $r-1$ objects are added into G. We claim that these inequalities imply

$$\sum_{j=1}^{n} H(\lfloor cov_j^1 + 1 \rfloor) c_j x_j \geq q.\theta \sum_{o_j \in G} c_j \tag{4}$$

*for the result set G returned by the greedy approach. Once (**??**) is proved, the theorem will follow by considering x be the incidence vector of an optimal feasible solution.*

*To prove (**??**), it is sufficient to exhibit nonnegative numbers $y_1, y_2, ..., y_m$ such that*

$$\sum_{i=1}^{m} p_{ij}y_i \leq H(\sum_{i=1}^{m} p_{ij})c_j \quad \text{for all } j \tag{5}$$

and such that

$$\sum_{i=1}^{m} y_i = \sum_{o_j \in G} c_j \tag{6}$$

for then

$$\begin{aligned} \sum_{j=1}^{n} H(\sum_{i=1}^{m} p_{ij}) c_j x_j & \geq \sum_{j=1}^{n} (\sum_{i=1}^{m} p_{ij} y_i) x_j \\ & = \sum_{i=1}^{m} (\sum_{j=1}^{n} p_{ij} x_j) y_i \\ & \geq q.\theta \sum_{i=1}^{m} y_i \\ & = q.\theta \sum_{o_j \in G} c_j \end{aligned}$$

as desired.

The numbers $y_1, y_2, ..., y_m$ satisfying (??) and (??) have a simple intuitive interpretation: each y_i can be interpreted as the cost distance achieved by MaxMargin for covering the keyword $q.\omega^i$. Without loss of generality, we can assume that G is $\{o_1,o_2,...,o_r\}$ when r objects are added into G, and so

$$\frac{cov_r^r}{c_r} \geq \frac{cov_j^r}{c_j}$$

for any r, j. If t objects are required to cover all query keywords, then

$$\sum_{o_j \in G} c_j = \sum_{j=1}^{t} c_j,$$

and

$$y_i = \sum_{r=1}^{t} \frac{c_r \cdot cw(o_r, q.\omega^i)}{cov_r^r}.$$

We know that

$$\sum_{i=1}^{m} y_i = \sum_{i=1}^{m}\sum_{r=1}^{t} \frac{c_r \cdot cw(o_r, q.\omega^i)}{cov_r^r} = \sum_{r=1}^{t} c_r$$

For any o_j, we know that cov_j^r decreases as the iteration continues from Lemma 5. We assume s is the largest superscript such that $cov_j^s > 0$ then

$$\begin{aligned}
\sum_{i=1}^{m} p_{ij} y_i &= \sum_{r=1}^{s} (cov_j^r - cov_j^{r+1}) \cdot \frac{c_r}{cov_r^r} \\
&\leq c_j \sum_{r=1}^{s} \frac{cov_j^r - cov_j^{r+1}}{cov_j^r} \\
&= c_j \sum_{r=1}^{s} \frac{cov_j^r - cov_j^{r+1}}{cov_j^r} \\
&\leq c_j \sum_{r=1}^{s} \frac{\lfloor cov_j^r + 1 \rfloor - \lfloor cov_j^{r+1} \rfloor}{\lfloor cov_j^r + 1 \rfloor} \\
&= c_j \sum_{r=1}^{s} \sum_{l=\lfloor cov_j^{r+1}+1 \rfloor}^{\lfloor cov_j^r + 1 \rfloor} \frac{1}{cov_j^r} \\
&\leq c_j \sum_{r=1}^{s} \sum_{l=\lfloor cov_j^{r+1}+1 \rfloor}^{\lfloor cov_j^r + 1 \rfloor} \frac{1}{l} \\
&= c_j H(\lfloor cov_j^1 + 1 \rfloor) - c_j H(\lfloor cov_j^s + 1 \rfloor) \\
&\leq c_j H(\lfloor cov_j^1 + 1 \rfloor)
\end{aligned}$$

Time Complexity. In general, the time complexity of MaxMargin consists of two components, namely the cost of building the initial feasible solution and the cost of searching the answer in Q. Suppose that the height of the LIR-tree is l. Thus, there are $n = \frac{|O|}{2^{l-1}}$ objects in each leaf node on average. As discussed earlier, the relevant objects in a nearest leaf node need to be organized in ascending order of their cost distances, and then added into FS. In the worst case, the time complexity of building the feasible solution is $O(nlogn)$. As with [?], we assume that there are m relevant objects to q. That is, the number of entries (i.e., the objects and nodes) in the priority queue Q is $O(m)$. Given that the entry e may be reinserted into Q in MaxMargin, in the worst case each entry e will be reinserted into Q at most $O(m)$ times. Thus, there is at most $O(m^2)$ iterations before the threshold constraint is satisfied. In each iteration, the major overhead comes from reinserting e into the ordered Q, which costs $O(logm)$. Therefore, the time complexity of searching the answer is m^2logm. As a result, the worst-case time complexity of MaxMargin is $O(nlogn + m^2logm)$.

6. Empirical Study

We conduct experiments to evaluate the performance of our proposed approaches using synthetic and real datasets. We specify the experimental setup in Section 6.1, and report the results on synthetic and real datasets in Sections 6.2 and 6.3, respectively.

Property	BT	GN
# of objects	298,346	977,302
# of keywords	748,162	5,286,498
# of distinct keywords	58,367	116,466

Table 3. Real dataset properties

Parameter	Range
DS (10^5)	0.1, **1**, 3, 5, 7, 9
TK	50, 100, 150, 200, 250, **300**
QK	2, **3**, 4, 5, 6, 7
TS	0.1, 0.2, **0.3**, 0.4, 0.5, 0.6
KD	3, **4**, 5, 6, 7, 8

Table 4. Parameter settings

6.1. Experimental Setup

Algorithms. In our study of the LCSK query processing, we implement the state-of-the-art approximate algorithm SUM-A used in [**?** **?**], and compare it with our proposed MaxMargin. To be fair, we consider the modified version of SUM-A based on the LIR-tree, which can better match our settings. In addition, we consider MergeList as a comparison for evaluating the effectiveness of approximate algorithms.

All algorithms were implemented in C/C++ and run in Windows 7 System on an Intel(R) Core(TM) i5-4590 CPU@3.30 GHz with 8GB RAM. Both KHT and LIR-tree are disk-resident, and the page size is set to 4 MB by default.

Datasets and queries. We deploy five synthetic and two real datasets in the experiments. Specifically, each synthetic dataset consists of three types of datasets following the uniform, random and zipf distribution. Table **??** presents the statistics of real datasets, namely BT [**?**] and GN [**?**]. BT is extracted from OpenStreetMap and all points fall into the rectangle [(-11.1, 49.6),(2.1,62.5)]. Each pair of values (e.g., (-11.1,49.6)) corresponds to a coordinate of the form *(latitude, longitude)*. The first coordinate corresponds to the bottom-left corner of the rectangle, and the second corresponds to the up-right corner. Each object in BT consists of a location and a set of keywords. GN is extracted from the U.S. Board on Geographic Names (geonames.usgs.gov). Similarly, each object is associated with a location and a textual description. We randomly generate the *level vector* and *cost* for the objects in the underlying dataset. Without loss of generality, we assume that the cost is in the range of $(0, 1)$.

As illustrated in Table **??**, we mainly focus on measuring the impact of five parameters: 1) *data size* (*DS*); 2) *the total number of distinct keywords in the spatial database* (*TK*); 3) *the number of keywords associated with each object* (*KD*); 4) *the number of query keywords* (*QK*); 5) *the threshold of q, i.e.,* $q.\theta$

(TS). All the default values are marked in bold in Table ??. Specifically, we study all these parameters on synthetic datasets and study QK and TS on real datasets since other parameters (e.g., TK) are fixed in real datasets.

For each workload, we randomly generate 20 queries based on the default values in Table ??. In the context of synthetic datasets, we generate the query location ($q.\ell$) and keywords ($q.\omega$) randomly for each query q. In the context of real datasets, however, a large number of the keywords are *rare*. That is, few objects contain them in their textual descriptions. Furthermore, these keywords are rarely considered as the query keywords by users. Motivated by these observations, we discard the keywords associated with at most 50 objects in the datasets from consideration. Then, we generate the query keywords randomly with the remaining keywords for each query q. The performance metrics that we measure are the response time and approximation ratio, and the results are computed by averaging the performance of 20 queries.

6.2. Results on Synthetic Datasets

We evaluate the overall performance of algorithms with three types of datasets, and report the average response time ($URZA\_T$) and the average approximation ratio ($URZA\_R$), where $URZA\_T = \frac{T_u+T_r+T_z}{3}, URZA\_R = \frac{R_u+R_r+R_z}{3}$. Here, Tu, Tr, Tz and Ru, Rr, Rz denote the response time and approximation ratio of uniform, random and zipf datasets, respectively.

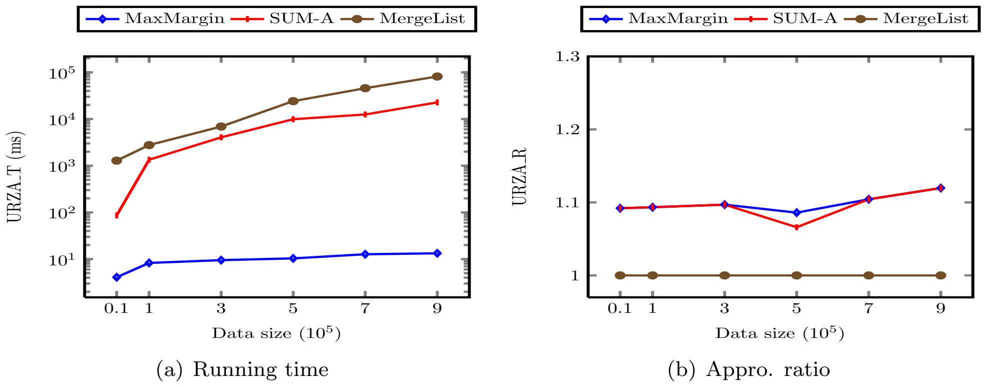
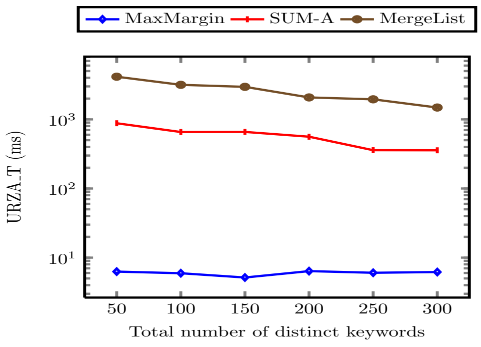


Fig. 10. Effect of DS on synthetic datasets

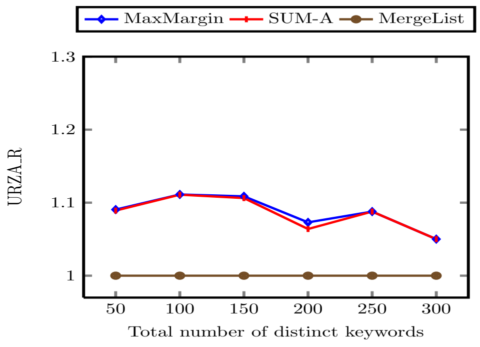
Varying the parameter DS. Fig. ??(a) shows that the response time of all algorithms increases as DS becomes larger, because much more relevant objects are required to be explored. Specifically, MaxMargin runs several orders of magnitude faster than SUM-A, which is largely due to the two pruning strategies employed by MaxMargin. We present the approximation ratio of algorithms in Fig. ??(b), and denote by the curve $y = 1$ the approximation ratio of MergeList. Even if both SUM-A and MaxMargin are based on the greedy strategy in [?], however, SUM-A achieves a little bit better performance than MaxMargin in terms of the accuracy. This might be due to the different strategies used to select the current optimal entry.

Varying the parameter TK. When TK varies from 50 to 300, the experimental results are shown in Fig. ??. As shown in Fig. ??(a), when TK increases, the response time of all algorithms decreases. The reason behind is that the number of relevant objects, namely $|RO_q|$, becomes smaller when TK becomes larger. As for the accuracy, we notice that SUM-A and MaxMargin have the same performance in most cases in Fig. ??(b), and both of them achieve good accuracy bounded by the upper bound 1.2.

Varying the parameter QK. The experimental results of algorithms when varying QK are shown in Fig. ??. As shown in Fig. ??(a), when QK increases, the response time of the two approximate algorithms increases slightly, whereas the performance of MergeList greatly degrades especially when QK varies from 3 to 5. Note that, we assign the default value of KD as 4 in Table ??. Thus, when QK is less than 4, all query keywords may be contained by a single object. In contrast, when QK is greater than 4, much more objects are required to be explored for satisfying the threshold constraint. Consistently, we observe that

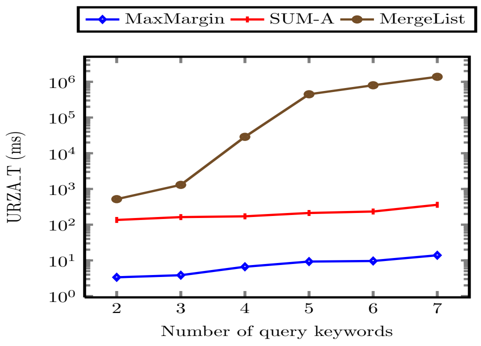


(a) Running time

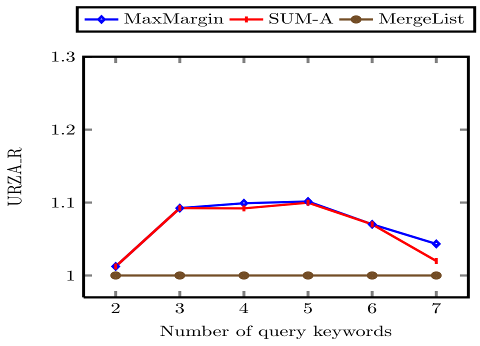


(b) Appro. ratio

Fig. 11. Effect of TK on synthetic datasets



(a) Running time



(b) Appro. ratio

Fig. 12. Effect of QK on synthetic datasets

the approximate algorithms achieve bad accuracy when QK varies from 3 to 5 (see Fig. **??**(b)). Also, we know that the first selected optimal entry by the greedy strategy dominates the subsequent selections, and thus has much effect on the accuracy. When much more objects are explored (when QK varies from 3 to 5), the first selected entry may not be optimal from the global view, which affects the whole performance of accuracy. Actually, this is an inherent limitation of the greedy strategy.

Varying the parameter TS. We measure the impact of TS on the performance of algorithms in this experiment, and present the results in Fig. **??**. As expected, the response time of all algorithms increases as TS increases. This is because much more objects would be explored before the threshold constraint is satisfied. Specifically, the response time of approximate algorithms increases almost linearly with TS. In contrast, MergeList increases rapidly. We observe that the accuracy of approximate algorithms fluctuates in Fig. **??**(b), denoting that TS has a greater impact on the accuracy of algorithms. This is because it determines the cardinality of the result set G.

Varying the parameter KD. As shown in Fig. **??**(a), the response time of all algorithms increases almost linearly with KD. The reason behind is that the response time is proportional to the number of relevant objects, i.e., $|RO_q|$, which is often proportional to KD. One might wonder why MergeList does not increase rapidly in this experiment. This is largely due to the two pruning strategies used by MergeList. We also notice that MaxMargin runs much faster than SUM-A, which demonstrates the effectiveness of our proposed two pruning strategies, namely the branch and bound strategy and the triggered update strategy. The results in Fig. **??**(b) show that both two approximate algorithms produce the near-optimal answers.

In the following, we delve more into details of the performance of algorithms on different types of datasets.

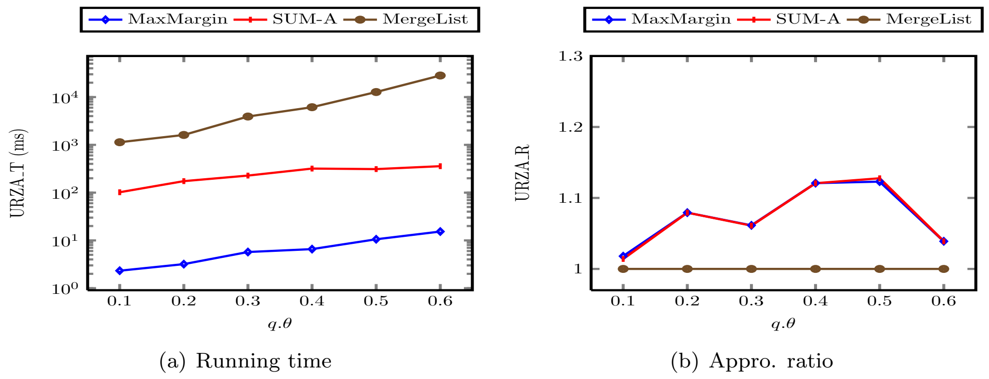


Fig. 13. Effect of *TS* on synthetic datasets

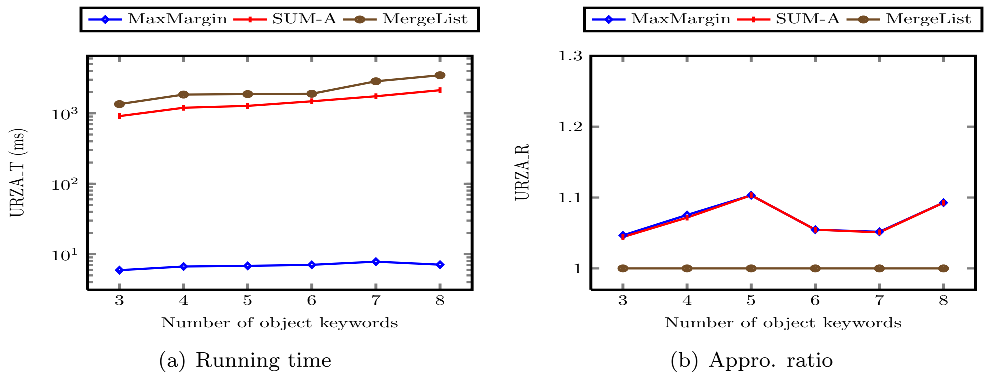


Fig. 14. Effect of *KD* on synthetic datasets

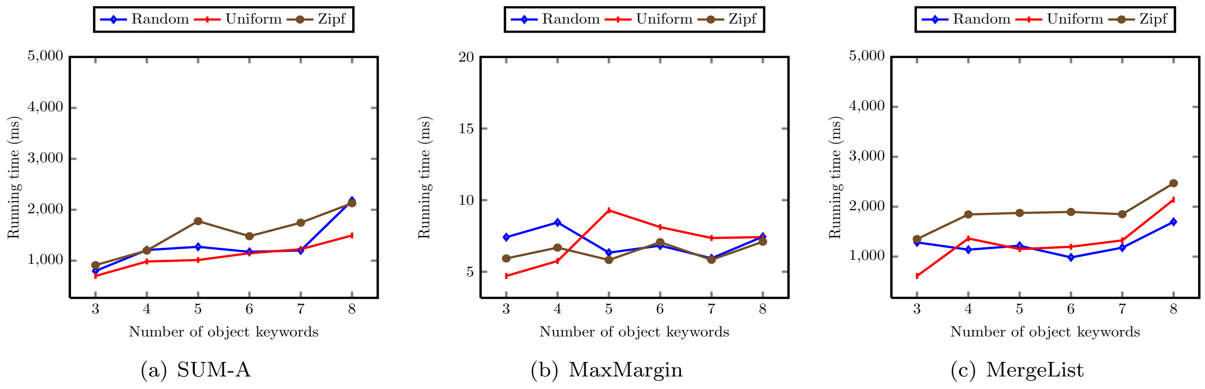
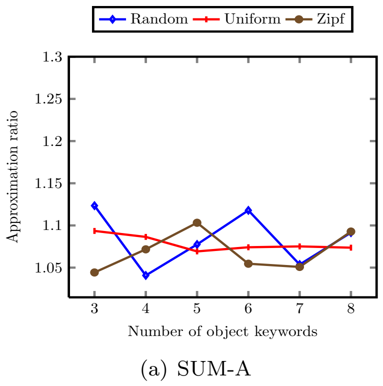
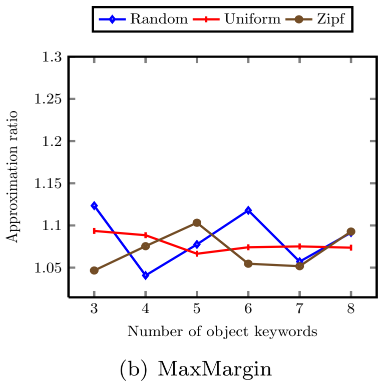


Fig. 15. Running time on different datasets

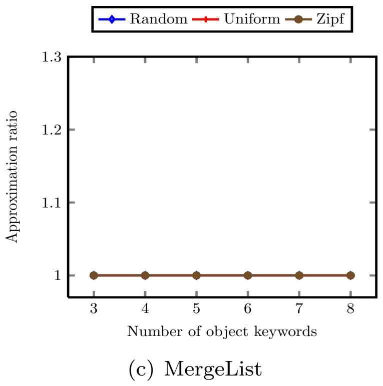
Figs. **??** and **??** present the response time and accuracy of algorithms with respect to *KD*. Specifically, we study the performance of algorithms on different types of datasets, namely uniform, random and zipf. It is shown that all algorithms are sensitive to zipf datasets. This is because the distribution of zipf datasets has much effect on the pruning power of algorithms. We observe that the uniform datasets achieve more stable accuracy. This is largely due to the distribution of the datasets. Fig. **??**(a) shows the LIR-tree index



(a) SUM-A

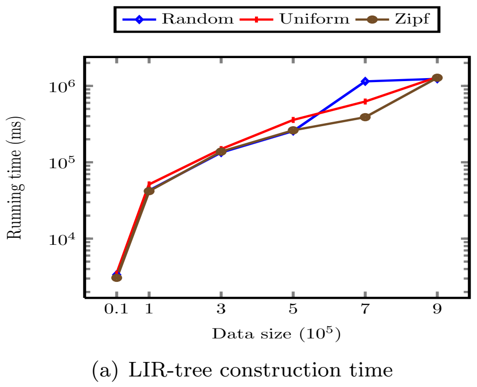


(b) MaxMargin

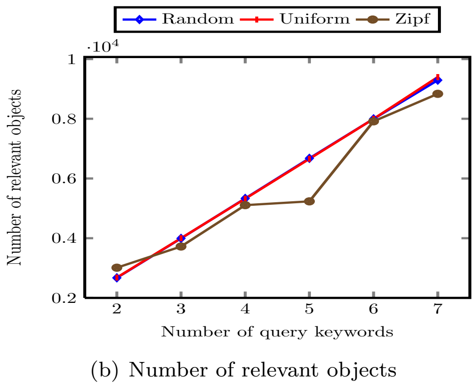


(c) MergeList

Fig. 16. Approximation ratio on different datasets



(a) LIR-tree construction time



(b) Number of relevant objects

Fig. 17. The performance on synthetic datasets

construction time with respect to *DS*. In general, the construction time increases almost linearly with *DS* for all three types of datasets. In addition, Fig. **??**(b) shows the number of relevant objects when *QK* increases. Similarly, the number of relevant objects is proportional to the number of query keywords.

6.3. Results on Real Datasets

We proceed to study the performance of algorithms on BT and GN. As noted earlier, other parameters (e.g., *TK*) are fixed in real datasets, thus we only evaluate the performance of algorithms with respect to *QK* and *TS*.

6.3.1. Evaluation on BT

Varying the parameter QK. As shown in Fig. **??**(a), MaxMargin and SUM-A outperform MergeList by several orders of magnitude in terms of the response time, especially when *QK* is greater than 3. We notice that, the average number of associated keywords with each object in BT is 3 (see Table **??**). When *QK* is greater than 3, MergeList may need to explore much more objects for satisfying the threshold constraint, which incurs prohibitive computation cost. In contrast, approximate algorithms prune the search space using LIR-tree. In real datasets, the number of keywords associated with each object is limited, which reduces the number of relevant objects and thus degrades the performance of pruning strategies used by MaxMargin. As shown in Fig. **??**(b), two approximate algorithms perform well in terms of accuracy. Specifically, the approximation ratio of them is close to 1, and they can produce the near-optimal answer in most cases.

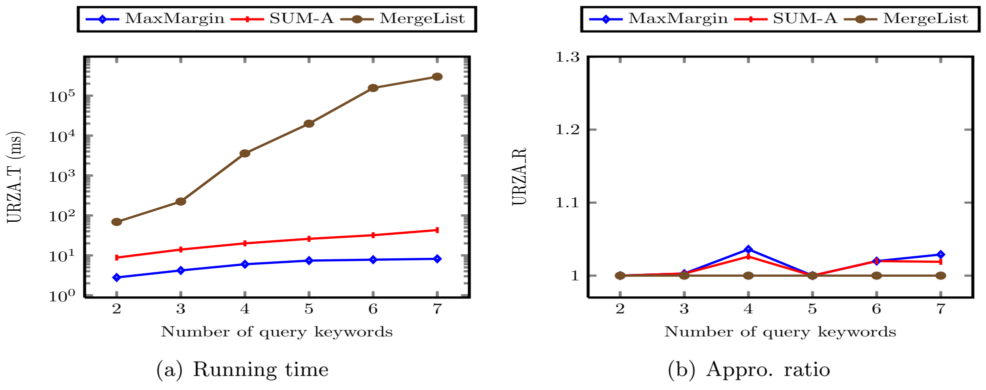


Fig. 18. Effect of QK on BT

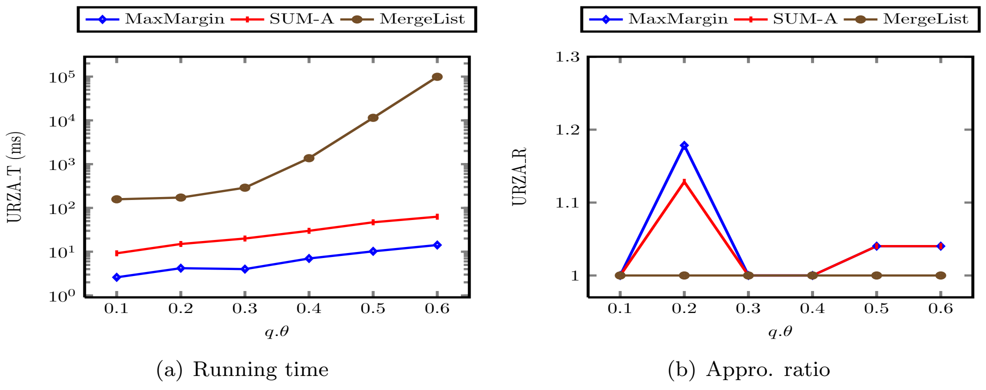


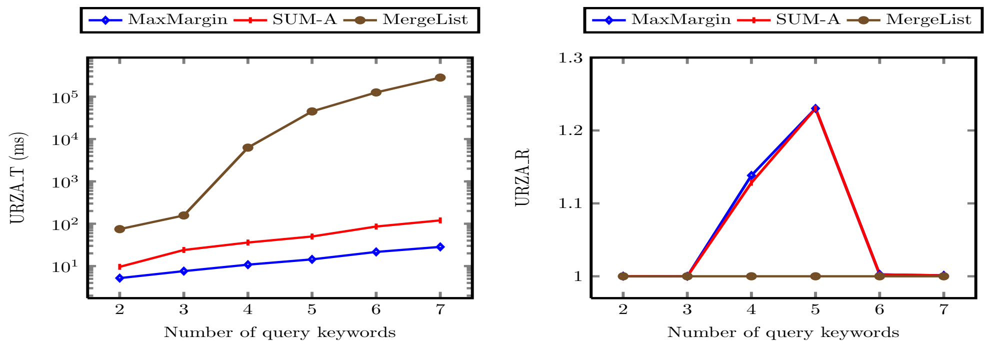
Fig. 19. Effect of TS on BT

Varying the parameter TS. As shown in Fig. **??**(a), approximate algorithms are reasonably efficient when TS varies from 0.1 to 0.6. However, the response time of MergeList increases dramatically. This is because much more objects in RO_q are required to be explored for finding the answer. With the help of LIR-tree, the response time of approximate algorithms increases almost linearly with TS. Fig. **??**(b) shows that, MaxMargin achieves the same accuracy as SUM-A in most cases, whereas its performance fluctuates. Specifically, when TS=0.2, the approximate algorithms achieve the worst accuracy. As noted earlier, there are five levels, and thus the weight for each level is 0.2 on average. When the threshold is greater than 0.2, more objects are required to satisfy the threshold constraint. This reveals the similar findings as with Fig. **??**(b), and thus can be explained with the similar reasons.

6.3.2. Evaluation on GN

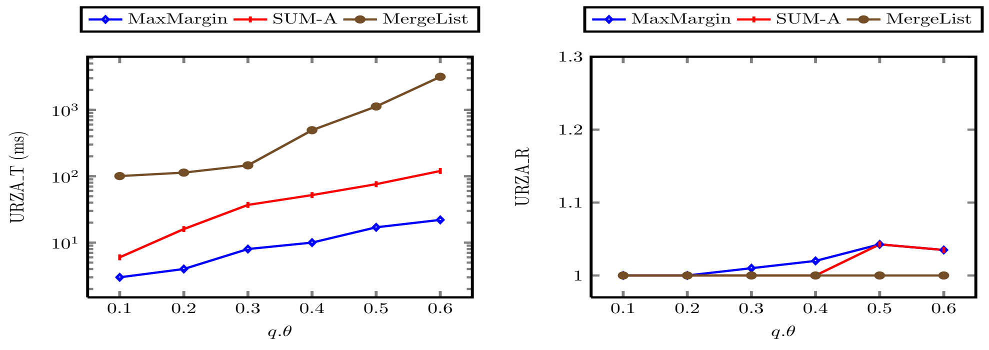
Varying the parameter QK. In this experiment, we study the performance of algorithms on GN by varying QK. As shown in Fig. **??**(a), the response time of approximate algorithms increases almost linearly with respect to QK, which is similar with the results achieved by BT. This demonstrates the effectiveness of LIR-tree. Also, we observe that all algorithms have much worse response time compared with BT, because the data size of GN is larger than BT. Fig. **??**(b) shows that the accuracy fluctuates when QK varies from 3 to 6. This is similar with the observation in Fig. **??**(b), and thus can be explained with the same reasons.

Varying the parameter TS. Fig. **??** shows the performance of algorithms as TS increases. Specifically, Fig. **??**(a) presents similar results with Fig. **??**(a). We observe that MergeList has better response time compared with the results of BT. The reason behind is that, with much more relevant objects in GN, it



(a) Running time (b) Appro. ratio

Fig. 20. Effect of QK on GN



(a) Running time (b) Appro. ratio

Fig. 21. Effect of TS on GN

is more convenient for MergeList to find objects with a greater coverage weight. Fig. **??**(b) shows that both two approximate algorithms achieve good accuracy, and return the near-optimal answer in most cases. Besides, the accuracy is more stable compared with that in Fig. **??**(b), it is probably because there are much more keywords associated with each object in GN compared with that in BT, and thus can conveniently find the desired object in each of the iterations from the global view.

7. Conclusions and Future Work

In this paper, we introduce a novel query type, called LCSK. LCSK can offer the desired result to users by taking into account the keyword level. We prove that the LCSK query is NP-hard by reducing the weighted cover set problem to it. We then propose two algorithms to answering this query exactly and approximately. The exact algorithm, namely MergeList, answers the query by searching the candidate space progressively with some pruning strategies, which is based on the flat index structure KHT. To be scalable to large datasets, we propose an approximate algorithm called MaxMargin. MaxMargin finds the answer by traversing the LIR-tree using the best-first strategy. Moreover, two pruning strategies, namely the branch and bound strategy and triggered update strategy are developed to improve the query performance. Extensive experiments on real and synthetic datasets are conducted to evaluate the performance of our proposed algorithms. As verified by the experiments that MaxMargin outperforms the state-of-the-art algorithm SUM-A by several orders of magnitude with the near-optimal answer.

There are several directions to do in the future. First, we can study the LCSK query in the dynamic environment with the moving or mobile objects. Then, it is also interesting to capture the importance of the objects using multiple feature vectors in the LCSK query.

Acknowledgements: We are grateful to anonymous reviewers for their constructive comments on this work. This work was supported in part by the National Science and Technology Supporting Plan-Project no. 2012BAH70F02 and 2013BAH62F01, and the cultural relic protection science and technology project of Zhejiang Province.