Collective Spatial Keyword Queries with Weight Coverage

Pengfei Zhang*, Huaizhong Lin, Dongming Lu

Department of Computer Science, Zhejiang University, China

Abstract

Spatial keyword queries have received significant attention recently since the proliferation of geo-textual data. Given a location and a set of query keywords, collective spatial keyword query (CoSKQ) returns a group of objects, which collectively cover the given query keywords and have the smallest cost. However, CoSKQ only retrieves objects that can cover query keywords, without taking the level information (e.g., the level of attraction), which is crucial for users to make decisions, of keyword into consideration.

To tackle the spatial keyword query with level information of keyword, keyword-aware group weight coverage query (KaGWC) is proposed in this work. KaGWC retrieves a group G of objects that collectively meet users' needs: 1) the weight that each query keyword covered by G not less than a given threshold; 2) the cost function is minimized. We map KaGWC query to the classic Weighted Set Cover (WSC) problem, denoting that it is NP-hard. With the KHT index, we design two approximation algorithms with provable approximation ratio and an exact algorithm. A comprehensive empirical evaluation over real and synthetic datasets demonstrates the efficiency and effectiveness of our proposed algorithms.

Keywords: Spatial database, collective spatial keyword query, level information, weight coverage

1. Introduction

With the development of information technology (e.g., GIS), as well as increasing popularity of services such as Google Earth and Baidu Lvyou, large volumes of spatial data (e.g., tourist attractions, hotels and restaurants) are becoming available. Recently, increasing focus is being given to geo-textual information in response to users' queries, which promotes the efforts on spatial keyword queries [8, 10, 21, 24]. By taking both of the locations and textual descriptions of content into consideration, spatial keyword queries offer greater flexibility to its users when looking for interesting objects.

A range of proposals have been published that aim to return relevant geo-textual objects in response to the user's personal preference. Such proposals could be classified into two categories according to result granularity: some return individual object [7, 10, 8, 19], while others return a group of objects [24, 25, 3, 15, 11]. In the first category, given a location and a set of keywords as arguments and returns the single spatial object that best matches these arguments. In a wide spectrum of applications, however, the users' needs (expressed by keywords) cannot be satisfied with only one single object. In the second category, mCK [24, 25], CoSKQ [3, 15] and BKC [11] retrieve a group of objects to satisfy users' needs collectively. However, few studies take into account the level information of keyword (e.g., the level of attraction, hotel and personal ability), which is crucial for users to make decisions.

For example, Table 1 presents a spatial objects database. Each keyword associates with an integer value, representing the corresponding level of attraction. In this situation, we assume that the larger the level the

*Corresponding author.

Email addresses: zpf_2013zb@zju.edu.cn (Pengfei Zhang), linhz@zju.edu.cn (Huaizhong Lin), ldm@zju.edu.cn (Dongming Lu)

[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.

OID	PX	PY	*Associated keywords*
o_1	159.0	246.0	mountain 4,landscape 1,temple 5
o_2	171.0	36.0	shore 2,museum 1
o_3	109.5	235.5	forest 4,mountain 1,temple 2
o_4	352.5	271.5	shore 1
o_5	97.5	276.0	driftage 1,shore 5,architecture 1
o_6	331.5	70.5	architecture 5,temple 2
o_7	259.5	177.0	museum 3,mountain 1,landscape 4
o_8	130.5	3.0	glacier 1
o_9	148.5	291.0	forest 4
o_{10}	204.0	58.5	driftage 3,mountain 1,glacier 1

Table 1: Spatial objects database

better the attraction. For users who prefer famous mountain may choose o_1, however, others may prefer to choose o_3, o_7 or o_{10} in light of the cost concerns or just for taking exercise. In such scenarios, users prefer to obtain objects that best match the personal preferences with level information of keyword, instead of retrieving objects only to cover the query keywords. To address this problem, we introduce the notion of **weight vector** to capture the weight users assign for each level to denote the preference. Without loss of generality, we hypothesize that the sum of each dimension of weight vector equals to 1. We aim at retrieving a group G of objects with minimum cost, and for each query keyword, the weight sum not less than a given threshold. By adjusting the threshold, users can control the size of result set flexibly and obtain the different succedaneous choices as the supplement. In addition, this query can also be used to find a consortium of partners that offer the required capabilities for a given project collectively. To accomplish a project, various kinds of capabilities are required, such as coding, information retrieval, text editing etc. Each partner with part of these capabilities, and we can measure the level of corresponding ability with an integer value. With the weight vector to capture the completion ratio for partners with different levels in a limited lifespan. In this situation, the leader may want to retrieve a consortium of partners, so that for each subtask e.g., coding, the total completion ratio not less than a threshold within a limited lifespan.

In our work, we enhance existing works from three aspects. First, in addition to geo-textual information considered by existing works, we also take into account level information of keyword, which is crucial for users to make decisions. Second, we introduce weight vector to capture the users' preferences, which offers greater flexibility to its users when looking for interesting objects. Third, different from existing CoSKQ search which takes the maximum sum cost, we consider the cost distance in [5] as our measurement. We believe this measurement is more realistic by considering both the cost of object and Euclidean distance between object o and query q. In view of this, we propose a new type of queries, namely, keyword-aware group weight coverage (KaGWC) query, which returns a group G of objects to meet users' needs collectively. Given these significant differences, the solution of KaGWC query can be very different from that of CoSKQ query.

Specifically, given a set of spatial objects O, and a query $q = (\ell, \theta, W, \omega)$, where ℓ is a spatial location and θ is a weight constraint threshold. W is a weight vector. ω represents the query keywords. KaGWC query retrieves a group G of objects that meet the following two conditions simultaneously:

- For each query keyword λ, the weight sum of G to λ is not less than θ;
- The cost distance of G is minimized.

Example 1: We illustrate an example of KaGWC query at the top of Table 2 with the objects presented in Table 1. We obtain all the feasible solutions (e.g., S_1 to S_{11}). Though all these solutions satisfy the weight constraint, we take S_1, the one with the minimum cost distance, as our optimal solution.

In this work, we map the classic WSC problem to KaGWC query, which denotes that KaGWC query is NP-hard. Due to the intrinsical challenges of answering KaGWC query, we design two efficient approximation algorithms, namely CubeTree and MaxMargin with provable approximation ratio. Considering the scale of

An example of KaGWC query			
$q.\ell$	$q.\theta$	$q.W$	$q.\omega$
(31.5,50.0)	0.4	(0.1,0.3,0.2,0.3,0.1)	mountain, temple

SID	Solutions	Coverage Weight	Cost Distance
S_1	o_1, o_3	0.4, 0.4	3746.83
S_2	o_1, o_3, o_6	0.4, 0.7	5851.73
S_3	o_1, o_3, o_7	0.5, 0.4	5834.71
S_4	o_1, o_3, o_{10}	0.5, 0.4	4610.38
S_5	o_1, o_6, o_7	0.4, 0.4	6530.98
S_6	o_1, o_6, o_{10}	0.4, 0.4	5306.65
S_7	o_1, o_3, o_6, o_7	0.5, 0.7	7939.61
S_8	o_1, o_3, o_6, o_{10}	0.5, 0.7	6715.28
S_9	o_1, o_3, o_7, o_{10}	0.6, 0.4	6698.26
S_{10}	o_1, o_6, o_7, o_{10}	0.5, 0.4	7394.53
S_{11}	$o_1, o_3, o_6, o_7, o_{10}$	0.6, 0.7	8803.15

Table 2: An example of KaGWC query

query keywords is limited, we also devise the exact algorithm MergeList in our work. All three algorithms are supported by inverted indexing the objects with keywords. CubeTree obtains the solution by taking the divide and conquer strategy in a bottom-up manner. Though this algorithm runs faster than other two algorithms, the performance is unstable in terms of approximation ratio. To overcome this critical drawback, we develop another approximation algorithm MaxMargin, which takes a greedy strategy. MaxMargin picks the current optimal object in each iteration until weight constraint is satisfied. MaxMargin runs slightly slowly than CubeTree, however, MaxMargin achieves better performance than CubeTree in terms of accuracy and stability of approximation ratio. Besides, we also devise the exact algorithm MergeList with two pruning strategies.

To summarize, we make the following contributions.

- We propose a novel type of queries, called KaGWC queries, retrieving a group of objects that cover the query keywords not less than a given threshold by taking into account the level information. We prove this problem is NP-hard. To the best of our knowledge, this is the first work to address this problem.
- We design two efficient and effective approximation algorithms based on the KHT index. Besides, we also propose an exact algorithm for this problem.
- We conduct comprehensive experiments to demonstrate the efficiency and effectiveness of our proposed algorithms.

The rest of this paper is organized as follows. Section 2 formally defines the problem and proves the NP-hard complexity of it. Section 3 discusses existing work. We present the index structure KHT and our proposed algorithms in Sections 4 and 5 respectively. Section 6 gives the empirical study and Section 7 concludes the paper.

2. Problem Statement

In this section, we first introduce the fundamental notions used in this paper. Then we will prove the NP-hard complexity of our problem. Notations used in this paper is summarized in Table 3.

Let O denotes a database containing n spatial objects. Each object $o \in O$ is associated with a location $o.\ell$, a set of keywords $o.\omega$ to capture the tourist attractions or hotels etc. and a $|q.\omega|$ dimensions vector $o.\nu$ with the ith element being the level information of ith keywords in $|o.\omega|$. For an object $o \in O$, we refer to the cost of o as cost(o). For ease of presentation, we take the level vector as a multidimensional positive

Notations	Explanations
q	The KaGWC query of form:$(\ell, \theta, W, \omega)$
RO_q	The relevant objects to query q
RO_λ	The relevant objects to keyword λ
$cost(o)$	The cost of o
$cd(o, q)$	The cost distance of o to query q
$cw(o, \lambda)$	The coverage weight of λ covered by o
$cov(G, q)$	The coverage weight of q covered by G
$cr(o, q)$	The contribution ratio of o to q
$Cube_\kappa$	The cube of keywords set κ
$dcr_q^r(o)$	The dynamic contribution ratio of o to q in iteration r

Table 3: Summary of the notations used.

integer vector. And the upper bound of integer value is fixed (e.g., for attraction not larger than 5, that is to say only has 5 levels for attraction).

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

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

In Equation(1), dist(o,q) refers to the Euclidean distance between o and q. Comparing with the cost function used in [3, 15], cost distance is more adaptive to real application scenarios in that, it not only take the distance between query point q and o but also the internal cost of o into consideration.

Definition 2. (Object coverage weight). Given a keyword λ, a multidimensional weight vector W and an object $o \in O$. We use the notation $o.\nu_\lambda$ to denotes the corresponding level value of keyword λ in $o.\nu$. Then the weight that λ covered by o can be represented as:

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

Differing with CoSKQ in which a keyword either covered by object o or not, in this work we take the coverage weight cw as our measurement. Note that if λ is not contained by $o.\omega$, we set $cw(o, \lambda) = 0$. We use $cov(o, q) = \sum_{\lambda \in q.\omega} cw(o, \lambda)$ and $cov(G, \lambda) = \sum_{o \in G} cw(o, \lambda)$ to denote the weight that q is covered by o and the weight that λ is covered by G, respectively.

Definition 3. (Contribution ratio). Given a query q and an object o, we define the contribution ratio of the object o to q as follows:

$$cr(o, q) = \frac{cov(o, q)}{cd(o, q)} \tag{3}$$

By taking both of the coverage weight and the cost distance into consideration, $cr(o, q)$ can fit in with the real application scenarios better than only use $cov(o, q)$ to evaluate the contribution of o to q.

Definition 4. (KaGWC queries). The KaGWC queries $q = (\ell, \theta, W, \omega)$, where ℓ is a spatial location and θ is a threshold. W is a vector model to capture level information and ω represents the query keywords. KaGWC queries aim at retrieving a group G of objects that collectively satisfy the following two conditions:

- For each keyword $\lambda \in q.\omega$, $cov(G, \lambda) \geq q.\theta$;
- $\arg\min_G \sum_{o \in G} cd(o, q)$.

Given a KaGWC queries q, we claim an object o is ***relevant*** to q if o contains at least one keyword $\lambda \in q.\omega$. We use notation RO_λ and RO_q to denote the set of objects that relevant to λ and query q in O, respectively. It is sufficient to take only RO_q instead of O into consideration for a specific query q. If a group G of objects can satisfy the weight constraint of definition 4, we say that G is a ***feasible solution*** of query q. Put differently, KaGWC queries return the feasible solution with minimum cost.

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

PROOF. *We can reduce the classic WSC Problem to the KaGWC problem. A typical instance of the WSC problem of the form* $< U, S >$*, where* $U = \{1, 2, 3, ..., n\}$ *of n elements and a family of sets* $S = \{S_1, S_2, S_3, ..., S_m\}$*, where* $S_i \subseteq U$ *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 F of S such that* $\cup_{S_i \in F} S_i = U$ *and such that its cost* $\sum_{S_i \in F} C_{S_i}$ *is minimized.*

To reduce the WSC problem to KaGWC queries q. We map each element of U corresponding to a query keyword, that each set S_i *corresponding to a spatial object* o_i*, and each positive cost* C_{S_i} *as the* $cd(o_i, q)$*. For each keyword associated with* o_i*, we set the level value to 1. Besides, we hypothesize the weight vector* $W = \{1, 0, 0, 0, 0\}$ *and the threshold is 1. Clearly, there is a solution to weighted set cover problem if and only if there is solution to query q.*

3. Related Work

In this section, we mainly overview the existing work related to our KaGWC queries, focusing mostly on conventional spatial keyword queries and collective spatial keyword queries.

3.1. Conventional Spatial Keyword Queries

The conventional spatial keyword queries [7, 10] take a location and a set of keywords as arguments, and return objects that can satisfy the users' needs solely. There are lots of efforts on conventional spatial keyword queries. By combining with existing queries in database community, there are several variants of conventional spatial keyword queries. We will review these queries as follows.

Combining with top-k queries. By combining with the top-k queries, the top-k spatial keyword queries [7, 12, 14, 18, 19, 21, 22] retrieve k objects with the highest ranking scores by utilizing the ranking function, which takes both location and the relevance of textual descriptions into consideration. To address the top-k spatial keyword queries efficiently, various hybrid indexes have been explored. This branch includes [7, 14] (IR-tree), [4] (SKI), [17, 18] (S2I). [8, 19] study the top-k spatial keyword queries over trajectory data. Cong et al. [8] utilizes the hybrid index B^{ck}-tree to facilitate the query processing of top-k trajectories. Wu et al. [21] handles the joint top-k spatial keyword queries utilizing the W-IR-Tree index. Zhang et al. [26] demonstrates that I^3 index, which adopts the Quadtree structure to hierarchically partition the data space into cells, is superior to IR-tree and S2I. Gao et al. [12] studies the reverse top-k boolean spatial keyword queries on the road network with count tree.

Combining with NN queries. The spatial keyword NN queries retrieve object that closes to the query location and contains the query keywords. Several variants have been explored. Tao et al. [20] proposes the new index called the SI-index to cope with multidimensional data, which overcomes the drawback of IR2-Tree [10] with Z-cuves. Lu et al. [16] studies the RSTkNN query, finding the objects that take a specified query object as one of their k most spatial-textual similar objects.

Combining with route queries. The conventional route queries [13] in spatial database search the shortest route that starts at location s, passes through as least one object from each category in C and ends at t. Yao et al. [23] proposes the multi-approximate-keyword routing(MARK) query, which searches for the route with the shortest length such that it covers at least one matching object per keyword with the similarity larger than the corresponding threshold value. The problem of keyword-aware optimal route search (KOR) is studied in [2], to find the route which can cover a set of user-specified keywords, a specified budget constraint is satisfied, and an objective score of the route is optimal. Three algorithms are proposed for this problem in [2], and the corresponding system of KOR is provided in their subsequent work [1].

Entry	Keywords	OList	FI	SI
e_1	$mountain$	o_1, o_3, o_7, o_{10}	5	0
e_2	$forest$	o_3, o_9	6	1
e_3	$landscape$	o_1, o_7	6	2
e_4	$shore$	o_2, o_4, o_5	6	1
e_5	$temple$	o_1, o_3, o_6	1	0
e_6	$museum$	o_2, o_7	2	0
e_7	$architecture$	o_5, o_6	5	1
e_8	$drigtate$	o_5, o_{10}	0	0
e_9	$glacier$	o_8, o_{10}	4	0

Table 4: KHT entries

3.2. Collective Keyword Queries

All these works aforementioned return object that can meet the users' needs solely. However, in real life applications, it is common to satisfy the users' needs collectively by a group of objects. The mCK queries [24, 25] return a set of objects to cover the query keywords. However, in the context of mCK, each object associates with only a single keyword and it only takes keywords into consideration. The most similar work to ours is CoSKQ querry [3, 15]. With the maximum sum cost function, Cao et al. [3] provides approximation algorithms as well as an exact algorithm. To further improve the performance, Long et al. [15] proposes a distance owner-driven approach, besides they also propose a new cost measurement called diameter cost and design an exact and an approximation algorithms for it.

Although both CoSKQ and our queries retrieve a group of objects as result and take the keywords and location into consideration, however, our work differs with CoSKQ mainly in three aspects: 1) we take the level information of keyword into consideration, which is crucial for users to make decisions; 2) we introduce weight vector to capture the users' preferences, which offers greater flexibility to its users when looking for interesting objects. 3) we take the combination of space distance and cost of object as our cost function, which is more closer to real life application scenarios. Due to these differences between CoSKQ and our KaGWC query, the methods adopted by CoSKQ can not be extended to solve our problem. Deng et al. [11] studies the BKC query which considers the keyword rating information and returns best keyword cover, which is different with ours in the query goal. Besides, BKC does not utilize the weight vector to capture the users' preferences.

4. KHT Index

As claimed in Section 2, given a query q it is sufficient to only tackle the relevant objects in RO_q, due to other objects have no contribution to satisfy the users' needs. To alleviate the unnecessary computation and boost the search efficiency, in this section, we introduce the keyword hash table (KHT) index, which is organized as a hash table with ***perfect hashing technique***.

In a nutshell, the KHT consists of two major components:

- Distinct keywords: A vocabulary of all the distinct keywords appearing in the object database.
- OID list: For each distinct keyword λ, there is a posting list which records the RO_λ.

Each entry of KHT of the form $(\lambda, \lambda.olist, FI, SI)$, where λ represents a keyword and $\lambda.olist$ is the RO_λ. FI and SI correspond to the first and second index of λ in KHT. Table 4 elaborates the KHT entries, and Fig. 1 shows the KHT structure of Table 1.

To balance the search efficiency and storage space, we combine the two-level index technique and perfect hashing technique [9] in our work. We assign each entry an index (Fi, Si), with which we can retrieve a KHT entry in nearly $\Theta(1)$.

As illustrated in Fig. 2, we determine the index of an entry in two steps. In the first step, we first map the keyword w of entry to the FTemp with the string hash function BKDRHash. It's worth to note that,

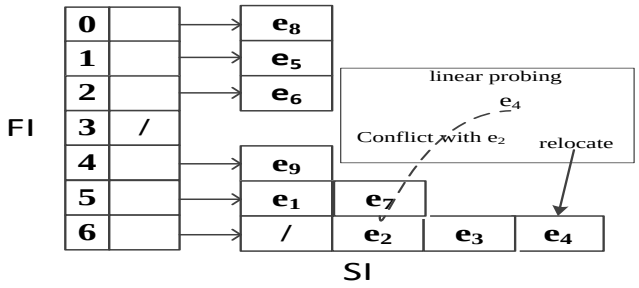


Figure 1: The KHT instance

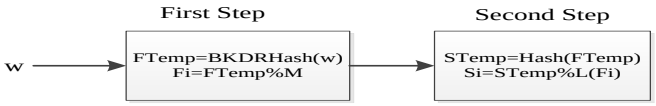


Figure 2: The Hash Function

although we apply BKDRHash as hash function, others string hash function can also be applied. And then, we obtain the Fi by performing a modulus on FTemp with first-level index length M. In the second step, we hash the FTemp with an integer hash function and then perform a modulus on STemp with second-level index length of Fi to get Si. After these two steps, each KHT entry has an index (Fi, Si). Note that, a delicate situation arises when two or more entries share the same index (Fi, Si) whose probability less than 0.5, which is demonstrated in [9]. In this case, ***"linear probing"*** technique is utilized to tackle this problem. From Table 4 we know that there is a conflict situation between "forest" and "shore". As illustrated in Fig. 1, when we try to insert the entry e_4 into KHT, e_2 has already stayed in location $(6, 1)$. Due to location $(6, 2)$ is occupied as well, with "linear probing", e_4 is inserted into $(6, 3)$ ultimately, just as the solid arrow shown.

With KHT, for a specific query q we can retrieve the RO_q in nearly $\Theta(|q.\omega|)$, and prune the unnecessary visits hugely.

5. Algorithms

Considering the inherent complexity of NP-hard problem, we design two approximation algorithms, namely CubeTree and MaxMargin with provable approximation ratio in Sections 5.1 and 5.2 respectively. Besides, we also propose the exact algorithm MergeList in Section 5.3.

5.1. Approximation Algorithm CubeTree

Similar to the divide and conquer strategy, CubeTree obtains the final solution by combining solutions of the sub-queries in a bottom-up manner. Before delving more into CubeTree algorithm, we introduce the notion of *keywords cube* which will lay the foundation for the CubeTree algorithm.

Definition 5. (keywords cube). Given a query q, a set of keywords $\kappa \subseteq q.\omega$ and $\kappa \neq \emptyset$. We define the keywords cube $Cube_\kappa$ as a subset of RO_κ, which is in the descending order of $cr(o, q)$ and satisfies the following three conditions:

- For each keyword $\lambda \in \kappa$, $cov(Cube_\kappa, \lambda) \geq q.\theta$;
- For any $o_i \in Cube_\kappa$, $o_j \in RO_\kappa - Cube_\kappa$, $cr(o_i, q) \geq cr(o_j, q)$;
- No subset of $Cube_\kappa$ satisfies the above two conditions.

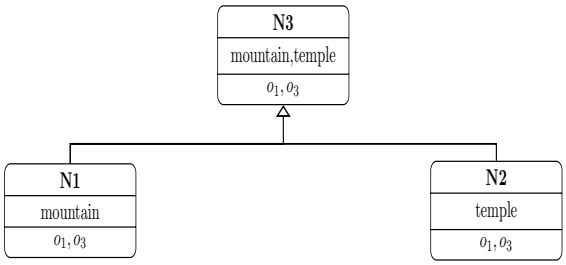


Figure 3: The cubeTree of query q in Table 2

For brevity, we refer to keywords cube as cube. Note that for an object o, it may exist in several cubes of different keywords set. In this paper hereafter, we use CubeTree to denote the algorithm and cubeTree to represent tree structure described below, whenever there is no ambiguity.

Each node of cubeTree of the form $(NID, \kappa, Cube_\kappa)$, where NID is the identifier of the cubeTree node. κ and $Cube_\kappa$ denote the keywords set and the corresponding cube, respectively. Fig. 3 depicts the cubeTree for query in Table 2.

In a nutshell, CubeTree algorithm builds the cubeTree iteratively in a bottom-up manner. For k query keywords, there are up to $2^k - 1$ nodes to be computed, which results in vast computation overhead. To alleviate the overhead, we adopt a strategy which can balance the overhead and the approximation bound. In this paper, instead of computing all of the nodes, we only compute a fraction of them as follows.

CubeTree iteratively constructs higher-level node by combining two adjacent low-level nodes, until there is only one single node in current level, which is the root node of cubeTree. In doing so, we only need to compute $\frac{k\cdot(k+1)}{2}$ nodes, which significantly reduces the computation overhead.

As illustrated in Fig. 3. In the bottom of cubeTree, there are two nodes corresponding to two query keywords, then we combine the adjacent two nodes, namely, N1 and N2 to form the higher-lever node N3. We return the cube of N3 as our approximation solution directly.

Lemma 1. *If* $\frac{cov(o_l,q)}{cd(o_l,q)} \geq \frac{cov(o_m,q)}{cd(o_m,q)} \geq \frac{cov(o_n,q)}{cd(o_n,q)}$*,then we have* $\frac{cov(o_l,q)+cov(o_m,q)}{cd(o_l,q)+cd(o_m,q)} \geq \frac{cov(o_n,q)}{cd(o_n,q)}$.

Proof. *It's obvious with the basic knowledge of fraction, we omit the proof here.*

With this lemma, we can construct the cubeTree with a simple strategy. In each iteration, we add the object with maximum $cr(o, q)$ into the cube of higher-level node instead of the object with maximum coverage weight.

We summarize our discussion above by three steps:

- Step 1 (Construct the bottom nodes): Construct the bottom nodes of cubeTree with KHT.
- Step 2 (Combine the low-level nodes): In this step, we obtain the higher-level node by combining two adjacent low-level nodes.
- Step 3 (Iterative step): Repeat Step 2 until there is only one single node in current level and we take this node as root. Then we return the cube of root as our final solution.

The pseudocode of CubeTree is outlined in Algorithm 1. CubeTree takes a bottom-up strategy to construct the cubeTree. CubeTree consists of two phases. In the first phase (lines 3-7), it builds the bottom nodes with KHT. Then, in the second phase (lines 8-19), by combining two adjacent low-level nodes (line 18), namely $cubeTree[lcID]$ and $cubeTree[rcID]$ to form a higher-level node $cubeTree[cID]$ in an iterative manner, until there is only one single node in this level, which is the root node. Finally, we return the cube of root as our final solution.

Algorithm 2 elaborates the procedure of the combination process. We use *oidl* and *oidr* to record the current optimal object of $cubeTree[lcID]$ and $cubeTree[rcID]$ respectively (lines 4-5). If $cr(oidl, q) \geq cr(oidr, q)$, we verify whether *oidl* contained by $cubecID$ or not (line 7). If *oidl* has been in $cubecID$, then

Algorithm 1: *CubeTree*

Input : The KHT and the query q.
Output: A group G of objects as approximation solution.

```
1  G ⟵ ∅;
2  lcID ⟵ 0; rcID ⟵ 0; cID ⟵ 0;
3  for i ⟵ 0 to |q.ω| − 1 do
4      cID ⟵ 2^i;
5      κ ⟵ q.ω[i];
6      construct the Cube_κ of keywords set κ with KHT;
7      put the node (cID, κ, Cube_κ) into cubeTree;
8  for i ⟵ 2 to |q.ω| do
9      for j ⟵ 0 to |q.ω| − i do
10         if j == 0 then
11             seq ⟵ 2^i − 1;
12         else
13             seq ⟵ seq × 2;
14         cID ⟵ seq;
15         lcID ⟵ seq − 2^(i+j−1);
16         rcID ⟵ seq − 2^j;
17         κ ⟵ cubeTree[lcID].κ ∪ cubeTree[rcID].κ;
18         cubecID ⟵ combineCube(lcID, rcID);
19         put the node (cID, κ, cubecID) into cubeTree;
20 solutionID ⟵ 2^|q·ω| − 1;
21 G ⟵ the cube of cubeTree[solutionID];
22 return G as the final solution;
```

we delete it from $cubeTree[lcID]$ and continue to select the next object. Otherwise, we add it into $cubecID$ and update the coverage weight of keywords in κ accordingly. Otherwise, if $cr(oidl, q) < cr(oidr, q)$, we tackle the *oidr* as the way of *oidl*. Finally, we return the cube of node cID to Algorithm 1.

Theorem 2. *The approximation ratio of CubeTree is not larger than* $\frac{cw_{max} \cdot cr_{max}}{cr_{min} \cdot q.\theta}$.

Proof. *Assuming that G is the solution returned by CubeTree. We use* $cw_{max} = max_{\lambda \in q.\omega} cov(G, \lambda)$ *to denote the maximum coverage weight of keyword in* $q.\omega$. *The maximum and minimum contribution ratio of object in G are* cr_{max} *and* cr_{min}, *respectively. We know that the cost of optimal solution:*

$$Cost(Opt) \geq \frac{|q.\omega| \cdot q.\theta}{cr_{max}} \tag{4}$$

and meanwhile,

$$Cost(G) \leq \frac{|q.\omega| \cdot cw_{max}}{cr_{min}} \tag{5}$$

holds. Combining inequalities (4) and (5), we know that

$$\frac{Cost(G)}{Cost(Opt)} \leq \frac{cw_{max} \cdot cr_{max}}{cr_{min} \cdot q.\theta}.$$

5.2. Approximation Algorithm MaxMargin

In section 5.1 we show how to utilize the CubeTree algorithm to handle our problem. Although CubeTree can solve our problem with provable approximation ratio, however, as presented in Theorem 2, the

Algorithm 2: *combineCube*

Input : Two low-level nodes lcID and rcID.
Output: The cube of the node cID.

```
1  cubecID ⟵ ∅;
2  κ ⟵ cubeTree[lcID].κ ∪ cubeTree[rcID].κ;
3  while exists λ ∈ κ and cov(cubecID, λ) < q.θ do
4      oidl ⟵ the object with maximum cr(o, q) in the cube of cubeTree[lcID];
5      oidr ⟵ the object with maximum cr(o, q) in the cube of cubeTree[rcID];
6      if cr(oidl, q) ≥ cr(oidr, q) then
7          if oidl ∈ cubecID then
8              delete oidl from the cube of cubeTree[lcID];
9              continue;
10         else
11             put oidl into cubecID;
12             delete oidl from the cube of cubeTree[lcID];
13             for each λ ∈ κ do
14                 update the cov(cubecID, λ);
15     else
16         if oidr ∈ cubecID then
17             delete oidr from the cube of cubeTree[rcID];
18             continue;
19         else
20             put oidr into the cubecID;
21             delete oidr from the cube of cubeTree[rcID];
22             for each λ ∈ κ do
23                 update the cov(cubecID, λ);
24 return cubecID;
```

approximation ratio of CubeTree is unstable due to the relevance with final solution. In this section, we elaborate the MaxMargin algorithm, which is inspired by the greedy strategy adopted by the WSC problem [6], which with more stable performance in terms of approximation ratio. To address WSC problem, the greedy strategy iteratively selects the current optimal subset and updates subsets that have not been visited yet accordingly. We modify this strategy by iteratively selecting the object with maximum $cr(o, q)$, and updating objects that have not been visited for our problem. The naive version of MaxMargin solves the problem by four steps. We use G to reserve the solution.

- Step 1 (Construct the RO_q): Construct the relevant set RO_q with KHT and sort the objects of RO_q in descending order of $cr(o, q)$.
- Step 2 (Select the optimal object): In this step, MaxMargin adds the current optimal object o with maximum $cr(o, q)$ into G. Then, o is deleted from RO_q.
- Step 3 (Update RO_q): After adding o into G, we update the $cr(o, q)$ of remaining objects in RO_q.
- Step 4 (Iterative step): Repeat Step 2 and Step 3 until the weight constraint in definition 4 is satisfied. Then we take G as our final solution.

Obviously, two major drawbacks degrade the performance of naive version: 1) lacking of efficient pruning strategies; 2) updating all the remaining objects for each iteration in Step 3 is time consuming. To further boost the search efficiency, we take a look at several notions which settle aforementioned drawbacks collectively.

Definition 6. (Keyword Max Priority Queue).: For a given keyword λ, we define the Keyword Max Priority Queue (KMPQ) of λ as the max priority queue of RO_λ according to the value of $cr(o, q)$.

Each element of KMPQ of the form (nid, oid, cr, cv), where nid and oid correspond to the identifier of element and the object, respectively. cr represents the contribution ratio of oid and cv is a $|q.\omega|$ dimensions vector to record the $cw(o, \lambda)$ for each keyword in $q.\omega$. Instead of constructing the RO_q, we construct the KMPQ for each query keyword. Once $cov(G, \lambda)$ reaches $q.\theta$, we can prune objects in the KMPQ of λ safely, which significantly exalts the performance.

To address the second drawback, we expand the notion of contribution ratio to fit in with our algorithm.

Definition 7. (Dynamic contribution ratio).: Given a query q and an object o, we define the dynamic contribution ratio (dcr) of object o in iteration r as follows:

$$dcr_q^r(o) = \frac{cov^r(o, q)}{cd(o, q)} \tag{6}$$

Note that the major difference between $dcr_q^r(o)$ and $cr(o, q)$ is that, we replace $cov(o, q)$ with $cov^r(o, q)$, where $cov^r(o, q)$ denotes the last contribution ratio of o after rth object added into result set. We do not update dcr for object o in each iteration r until o is chosen as the current optimal object.

Lemma 2. *Given a query q, an object o, two integers m, n and $m \leq n$, then $dcr_q^n(o) \leq dcr_q^m(o)$.*

PROOF. *After each iteration r, the value of $cov^r(o, q)$ is comparable or below than the former iteration. Besides, $cd(o, q)$ is constant in each iteration, which results in the decreasing of $dcr_q^r(o)$. If $m \leq n$, we can safely draw the conclusion that the $dcr_q^n(o)$ is not larger than $dcr_q^m(o)$.*

Instead of updating all the remaining objects after each iteration, we update the object until it is chosen as the optimal object. Lemma 2 indicates that we can always choose the newly updated object with maximum $dcr_q^r(o)$ as current optimal object, and objects whose dcr less than $dcr_q^r(o)$ unnecessary to be updated, which significantly prunes the updating overhead.

We summarize the aforementioned discussion by following three steps to improve the naive version. And we use G to reserve the final solution and RV to record the residuals.

- Step 1 (Construct the KMPQ): In this step, we construct the KMPQ for each keyword of $q.\omega$.
- Step 2 (Select the optimal object): Select the current optimal object o from all the KMPQs. If o already in G, then continue to select next optimal object. Otherwise, we add o into G if $cov^r(o, q)$ less than RV in all dimensions. Otherwise, we update the $cr(o, q)$ of o and reinsert it into KMPQ.
- Step 3 (Iterative step): Repeat Step 2 until RV equals to 0 in all dimensions. Then we take G as our final solution.

Algorithm 3 illustrates the details of MaxMargin algorithm.

MaxMargin utilizes the residual vector RV to record the difference between $q.\theta$ and the coverage weight of G for each query keyword dynamically. Initially, each dimension of RV is set to be $q.\theta$ (line 3). There are two different cases when processing the selected optimal object in each iteration. Case 1: If all dimensions satisfy $maxOID.cv[i] < RV[i]$ (lines 16-25), it needs not to update maxOID and we update RV accordingly. Case 2: Otherwise, we update the dcr of $maxOID$ and the PQ[indicator]. Algorithm terminates until RV equals to 0. Note that, in Algorithm 3, we set $PQ[j]$ to invalid if $RV[j]$ is 0 (line 23). Put differently, we prune all the objects in $PQ[j]$ once the $cov(G, q.\omega[j])$ reaches the threshold $q.\theta$. What's more, we update the object $maxOID$ (lines 27-32) only when the coverage weight of o larger than RV in some dimensions instead of updating the remaining RO_q in naive version. Both of these two strategies significantly improve the performance of MaxMargin algorithm.

Example 2: Let's go back to the KaGWC query q depicted in Table 2. As illustrated in Fig. 4, we answer the query q in four steps, as follows.

- Step 1 (Initial state): In this step, we construct the KMPQ for "mountain" and "temple", and initiate the $RV = \{0.4, 0.4\}$ and $G = \emptyset$ as depicted in Figs. 4a and 4b.

Algorithm 3: *MaxMargin*

Input : The KHT and the query q.
Output: A group G of objects as approximation solution.

```
1  G ⟵ ∅; maxMargin ⟵ 0.0; indicator ⟵ −1;
2  for i ⟵ 0 to |q.ω| − 1 do
3      RV[i] ⟵ q.θ;
4      construct the KMPQ PQ[i] for q.ω[i] with KHT;
5  r ⟵ 0;
6  while exists a dimension i satisfies RV[i] > 0.0 do
7      for each valid PQ[j] ∈ PQ do
8          o ⟵ PQ[j].top();
9          if dcr_q^r(o) > maxMargin then
10             maxMargin ⟵ dcr_q^r(o);
11             indicator ⟵ j;
12             maxOID ⟵ o;
13     if maxOID ∈ G then
14         PQ[indicator].dequeue();
15         continue;
16     if all dimensions satisfy maxOID.cv[i] < RV[i] then
17         G ⟵ G ∪ {maxOID};
18         for j ⟵ 0 to |q.ω| − 1 do
19             if RV[j] ≥ maxOID.cv[j] then
20                 RV[j] ⟵ RV[j] − maxOID.cv[j];
21             else
22                 RV[j] ⟵ 0;
23                 set the PQ[j] to be invalid;
24         PQ[indicator].dequeue();
25         r++;
26     else
27         for j ⟵ 0 to |q.ω| − 1 do
28             if RV[j] < maxOID.cv[j] then
29                 maxOID.cv[j] ⟵ RV[j];
30         recompute the dcr_q^r(maxOID);
31         PQ[indicator].dequeue();
32         PQ[indicator].enqueue(maxOID);
33 return G as the final solution;
```

- Step 2 (Fetch o_3): Due to the two KMPQs have common optimal object o_3, we can select o_3 from any of these two KMPQs. And we select o_3 from "mountain". After fetching o_3 from "moutain", the KMPQ of "moutain" is reorganized and the RV and G are updated as Fig. 4c.
- Step 3 (Delete o_3): As depicted in Figs. 4c and 4d. After fetching o_3, the current optimal object is o_3 of "temple". However, o_3 has been in G. So we delete it from "temple" directly, as illustrated in Fig. 4f.
- Step 4 (Fetch o_1): Again, the two KMPQs have common optimal object. We select o_1 from "mountain", and put it into G. Up to now, RV is equal to 0. We terminate our procedure and return G as the final solution.

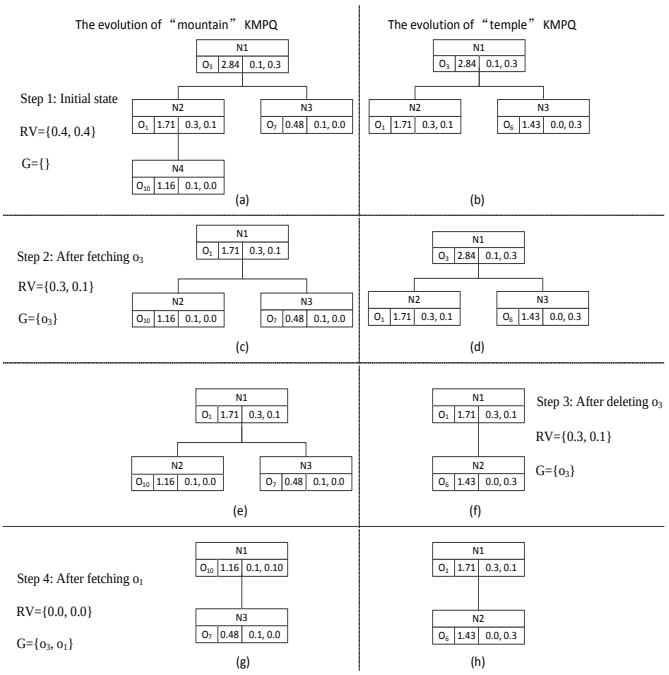


Figure 4: KMPQ of query q in Table 2

Theorem 3. *The approximation ratio of MaxMargin is not larger than* $\frac{H(\lfloor cov+1 \rfloor)}{q.\theta}$*, where cov is the largest* $cov(o_j, q)$ *for all* $o_j \in RO_q$*.*

PROOF. *Inspired by the proof in [6], we provide the approximation ratio proof of MaxMargin here. We use m, n to denote the number of elements in* $|q.\omega|$ *and* $|RO_q|$ *respectively. We define a* $m \times n$ *matrix* $P = (p_{ij})$ *by*

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

According to the definition of P, we know that the n columns of P is n coverage weight vectors of n objects. The goal of MaxMargin is to retrieve a group G of objects. And we utilize the incidence vector $x = (x_j)$ *to denote the cover set. Clearly, the incidence vector x of an arbitrary cover satisfies:*

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

For ease of presentation, in the following, we refer to the cost distance of o_j *as* c_j*. And we claim that these inequations imply*

$$\sum_{j=1}^{n} H(\lfloor cov_j^1 + 1 \rfloor) c_j x_j \geq q.\theta \sum (c_j : where\ o_j \in G) \tag{7}$$

for the cover G returned by the greedy heuristic. Once (7) is proved, the theorem will follow by letting x be the incidence vector of an optimal cover.

To prove (7), it will suffice to exhibit nonnegative numbers $y_1, y_2, ..., y_m$ such that

$$\sum_{i=1}^{m} p_{ij} y_i \leq H(\sum_{i=1}^{m} p_{ij}) c_j \quad for\ all\ j \tag{8}$$

and such that

$$\sum_{i=1}^{m} y_i = \sum (c_j : where\ o_j \in G) \tag{9}$$

for then

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

as desired.

The numbers $y_1, y_2, ..., y_m$ satisfying (8) and (9) have a simple intuitive interpretation: each y_i can be interpreted as the cost paid by MaxMargin for covering the keyword $q.\omega[i]$. We use cov_j^r to denote the coverage weight of query q covered by object o_j at the beginning of iteration r. Without loss of generality, we may assume that G is $\{o_1, o_2, ..., o_r\}$ after r iteration, and so

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

for all r and j. If there are t iterations altogether then

$$\sum (c_j : j \in G) = \sum_{j=1}^{t} c_j,$$

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 in RO_q, we know that the cov_j^r decrease as the iteration continues. We assume s is the largest superscript such that $cov_j^s > 0$ then

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

$$
\begin{aligned}
&= c_j \sum_{r=1}^{s} \frac{cov_j^r - cov_j^{r+1}}{cov_j^r} \\
&\leq c_j \sum_{r=1}^{s} \frac{\lfloor cov_j^r + 1 \rfloor - \lfloor cov_j^{r+1} \rfloor}{\lfloor cov_j^r + 1 \rfloor} \\
&= c_j \sum_{r=1}^{s} \sum_{l=\lfloor cov_j^{r+1}+1 \rfloor}^{\lfloor cov_j^r+1 \rfloor} \frac{1}{cov_j^r} \\
&\leq c_j \sum_{r=1}^{s} \sum_{l=\lfloor cov_j^{r+1}+1 \rfloor}^{\lfloor cov_j^r+1 \rfloor} \frac{1}{l} \\
&= c_j H(\lfloor cov_j^1 + 1 \rfloor) - c_j H(\lfloor cov_j^s + 1 \rfloor) \\
&\leq c_j H(\lfloor cov_j^1 + 1 \rfloor)
\end{aligned}
$$

5.3. The Exact Algorithm MergeList

For many real application scenarios, the number of query keywords submitted by users is limited. Motivated by this observation, we devise an exact algorithm MergeList in this section.

The straightforward method for the exact algorithm is to enumerate all the subsets of RO_q and return the subset, which covers the query keywords not less than a given threshold and with minimum cost, as our optimal solution. This yields an exponential time complexity in terms of the number of objects in RO_q.

To further prune the search space, we delve into several efficient pruning strategies below.

Firstly, we sort the objects of RO_q in ascending order of the cost distance. And we record the current optimal solution and its cost with notations COS and $minCost$. Instead of enumerating all the subsets of RO_q randomly, we construct the candidate subsets whose cost less than minCost by adding object into existing candidate subsets progressively.

Lemma 3. *Given a sorted RO_q which in ascending order of the cost distance. If for each existing candidate set **ecs**, the cost sum of ecs and the current visitorial object **cvo** in RO_q is not less than minCost, then COS is the optimal solution.*

PROOF. *Since RO_q is sorted in ascending order of the cost distance. We know that each object behind cvo in RO_q has a higher cost than cvo. If the cost sum of any ecs and cvo is not less than minCost, we can conclude that any ecs can not be the optimal solution, so we can terminate our procedure immediately.*

Secondly, there is an ***"Apriori property"*** in the data mining field. ***"All nonempty subsets of a frequent itemset must also be frequent"***. In the sequel, we present a similar pruning strategy.

Lemma 4. *If the cost sum of an ecs and cvo larger than the minCost, we can prune the ecs safely and any superset of it needs not to be computed.*

PROOF. *If the cost sum of an ecs and cvo larger than the minCost, we know that ecs cannot be the optimal solution and any superset of ecs neither can be the optimal solution as well, so we can prune them safely.*

Further, only if G is a feasible solution, we can filter out any superset G', since G is superior to G' anyway.

In a word, Lemma 3 allows us to terminate the procedure earlier and Lemma 4 provides significant pruning ability. We elaborate the MergeList in Algorithm 4.

The MergeList algorithm can be summarized as following three steps.

- Step 1 (Construct RO_q): In this step, we construct the RO_q and sort it in ascending order of cost distance.

- Step 2 (Add o into *ecs*): For object o in RO_q, we verify for each ecs whether delete it from candidate sets SS or combine it with o and put it into SS.
- Step 3 (Iterative step): Repeat Step 2, until the terminal condition in Lemma 3 is met.

Algorithm 4: *MergeList*

Input : The KHT and the query q.
Output: A group G of objects as resulting solution.

```
1  COS ⟵ ∅;
2  SS ⟵ ∅;
3  minCost ⟵ INFINITE_MAX;
4  RO_q ⟵ compute the relevant object set to query q with KHT;
5  sort RO_q in ascending order of weight distance cost;
6  put empty set ∅ into SS;
7  for each object o ∈ RO_q do
8      if wd(o, q) ≥ minCost then
9          break;
10     for each ecs ∈ SS do
11         if the cost sum of ecs and o not less than minCost then
12             delete ecs from SS;
13             continue;
14         tempSet ⟵ ecs ∪ {o};
15         if tempSet is a feasible solution then
16             COS ⟵ tempSet;
17             minCost ⟵ the cost of tempSet;
18             delete ecs from SS;
19         else
20             put tempSet into SS;
21     if SS == ∅ then
22         break;
23 G ⟵ COS;
24 return G as the final solution;
```

As discussed above, in MergeList algorithm, we construct the candidate set by adding the object into ecs progressively. We use SS to store the ecs, and initiate SS with the empty set (line 6). For each object o in RO_q, if the cost of o not less than minCost then we terminate our procedure (lines 8-9). Otherwise, for each ecs satisfies Lemma 4, we prune it directly (lines 11-13). If *tempSet* is a feasible solution, we use it to update the COS and *minCost* (lines 15-18), otherwise we add it into SS.

Example 3: Consider a query q with two keywords $q.\omega = \{\lambda_1, \lambda_2\}$. Fig. 5a illustrates the RO_q and the cost distance to q and the corresponding coverage weight. We illustrate the process of MergeList in Fig. 5b. There are total six steps to answer this query.

- Step 1 (Initiation): We initiate the SS, *minCost* and COS in this step.
- Step 2 (Visit o_1): Due to RO_q in Fig. 5a has been sorted, we visit the object according to the order in Fig. 5a We first visit o_1, and merge it with ecs in SS.
- Step 3 (Visit o_2): Object o_2 is merged with ecs in SS.
- Step 4 (Visit o_3): In this step, we obtain a feasible solution $COS = \{o_2, o_3\}$. And to filter candidate sets with Lemma 4.
- Step 5 (Visit o_4): In this step, ecs which satisfies Lemma 4 is pruned by COS.

OID	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)

SID	Action	SS	minCost	COS
S_0	Initiation	$\emptyset$	$+\infty$	$\emptyset$
S_1	Visit o_1	$\emptyset$,$\{o_1\}$	$+\infty$	$\emptyset$
S_2	Visit o_2	$\emptyset$,$\{o_1\}$,$\{o_2\}$,$\{o_1,o_2\}$	$+\infty$	$\emptyset$
S_3	Visit o_3	$\emptyset$,$\{o_1\}$,$\{o_2\}$,$\{o_1,o_2\}$,$\{o_3\}$,$\{o_1,o_3\}$	6.5	$\{o_2,o_3\}$
S_4	Visit o_4	$\emptyset$,$\{o_4\}$	6.5	$\{o_2,o_3\}$
S_5	Visit o_5	$\emptyset$,$\{o_4\}$	6.5	$\{o_2,o_3\}$

(b)

Figure 5: An example of MergeList

- Step 6 (Visit o_5): Because the cost of o_7 is larger than *minCost*, so Lemma 3 is met and COS is the optimal solution.

Theorem 4. *(Correctness of MergeList): The MergeList algorithm always produce the correct result set.*

PROOF. *Assuming the number of objects in RO_q is n, hence, there are up to $2^n - 1$ candidate sets. Each ecs either used to update the COS or pruned by the COS. If the cost of ecs less than minCost, we take ecs as our COS. Otherwise, we prune ecs and any superset of it. Hence, it is suffice to show that MergeList never prune any feasible solution whose cost less than minCost (false negatives), and never maintain the ecs whose cost larger than minCost (false positives). So the MergeList algorithm always produce the correct result.*

6. Empirical Study

In this section, we experimentally study the performance of our proposed algorithms through comprehensive experiments on both real and synthetic datasets. We describe the experimental settings in Section 6.1, and report the performance of our proposed algorithms for KaGWC queries on synthetic and real datasets in Sections 6.2 and 6.3 respectively.

6.1. Experimental Setup

Algorithms. We evaluate the performance of the following proposed algorithms: the approximation algorithm CubeTree in Section 5.1, the approximation algorithm MaxMargin in Section 5.2, and the exact algorithm MergeList in Section 5.3.

Data and queries. We conduct experiments with two datasets, namely synthetic dataset and real-life dataset. Our experiments were carried out primarily on synthetic dataset with five major factors, namely, 1) *the total number of keywords in the spatial object database* (TK), 2) *data size* (DS), 3) *the number of query keywords* (QK), 4) *the upper bound of the number of keywords associated with each object* (KD), not the fixed number of keywords in [24], 5) *the weight constraint threshold of query q* (TS). We study their effect on both response time and approximation ratio in three algorithms. Synthetic data generator generates three types of datasets following the uniform, random and zipf (URZ) distribution respectively. For each query, we repeat 5 times experiment for each type of dataset. We also included real dataset CA, which was collected from the U.S. Board on Geographic Names(geonames.usgs.gov). For each query, we repeat 10 times experiment.

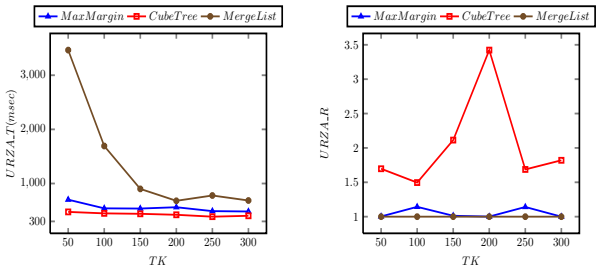
All three algorithms were implemented in C/C++ and run on an Intel(R) Core(TM)2 Quad CPU Q8400 @2.66Hz with 4GB RAM.

6.2. Results on Synthetic Dataset

To measure the comprehensive performance of our algorithms on different types of datasets, in this section, we employ the URZ average response time ($URZA\_T$) and URZ average approximation ratio ($URZA\_R$) as the measurements, where $URZA\_T = \frac{T_u+T_r+T_z}{3}, URZA\_R = \frac{R_u+R_r+R_z}{3}$. Tu, Tr, Tz and Ru, Rr, Rz represent the response time and approximation ratio of uniform, random and zipf distribution datasets, respectively. Table 5 illustrates all the possible values for these factors in subsequent experiments. Note that, we use bold font to denote the default value for each factor. We repeat query 5 times for each dataset, and we take the average value of all datasets as the final result.

Factors	Instance Value
TK	50,100,150,200,250,**300**
DS(10^4)	1,**10**,30,50,70,90
QK	2,**3**,4,5,6,7,8,9
KD	3,5,**7**,9,11,13,15
TS	0.1,**0.2**,0.3,0.4,0.5,0.6,0.7

Table 5: The real data of CA

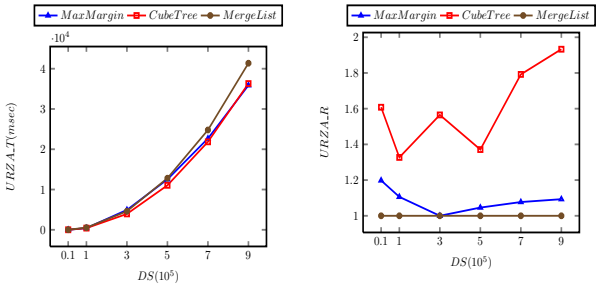


(a) (b)

Figure 6: Varying the factor TK

Varying the factor TK. Figs. 6a and 6b show the response time and approximation ratio of three algorithms respectively, when we vary the value of TK. As shown in Fig. 6a, the response time of MergeList decreases dramatically as TK becomes larger. CubeTree runs faster than does MaxMargin. And both CubeTree and MaxMargin decrease slightly as TK is increased. The reason behind is that, when other factors are fixed, the larger the TK, the smaller number of relevant objects, and thus smaller objects need to be handled during algorithms execution. It is also denoted that MergeList is more sensitive than CubeTree and MaxMargin in terms of TK. In Fig. 6b, we set the approximation ratio of the exact algorithm MergeList equals to 1, which is also adopted by the subsequent experiments. It can be observed from Fig. 6b that, MaxMargin achieves a much better accuracy than CubeTree. And the approximation ratio of MaxMargin changes slightly and with an upper bound 1.25 when we vary the value of TK. In contrast, the approximation ratio of CubeTree changes dramatically and with an upper bound 3.5. This is because MaxMargin always select the optimal object and update the contribution ratio dynamically.

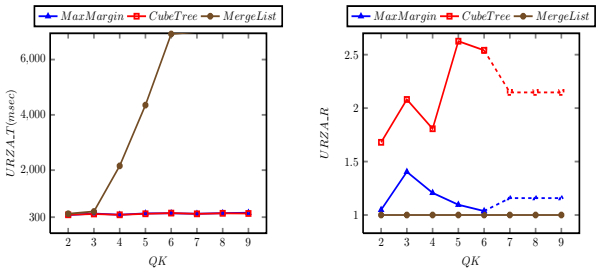
Varying the factor DS. Figs. 7a and 7b show the response time and approximation ratio of three algorithms respectively, when we vary the value of DS. We vary the value of DS from 10,000 to 900,000. As expected, as DS increases, the response time of all three algorithms increase. However, as shown in the Fig. 7a, the response time of MergeList does not increase dramatically as DS is increased. This is due to the two pruning strategies Lemma 3 and Lemma 4 employed in MergeList. Specifically, although CubeTree runs faster than MaxMargin but the response time of MaxMargin increases slowly than CubeTree in that the merge phase takes much time when DS increases, which reflects that MaxMargin is more adaptive to DS than CubeTree. Fig. 7b shows MaxMargin achieves an approximation ratio from 1.0 to 1.2 when we



(a) (b)

Figure 7: Varying the factor DS

vary DS from 10,000 to 900,000. Especially, the approximation ratio equals to 1 when DS takes the value of 300,000. The approximation ratio of CubeTree with an upper bound 2.0, however, it is unstable.

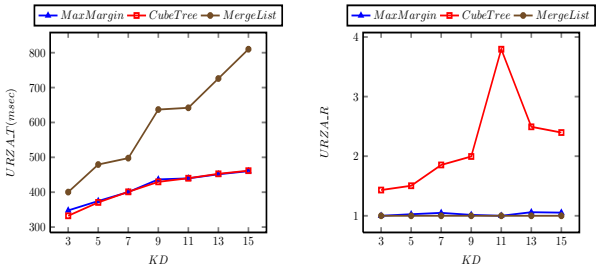


(a) (b)

Figure 8: Varying the factor QK

Varying the factor QK. Figs. 8a and 8b show the response time and approximation ratio of three algorithms respectively, when we vary the value of QK. In this experiment, we vary the value of QK from 2 to 9. We omit the experiment results when MergeList runs out of memory (e.g., we omit the response time of MergeList in Fig. 8a when QK takes the value of 7, 8 and 9, in that it runs out of memory). As shown in Fig. 8a, MaxMargin and CubeTree run faster than MergeList. Usually, they are 7-20 times faster than MergeList and their response times are almost not affected by QK as shown in Fig. 8a. This is because that with the KHT index, CubeTree can merge the cubeTree nodes efficiently and MaxMargin can select the optimal object from KMPQ directly. Since the exact algorithm MergeList fails to return the optimal solution when QK takes value of 7, 8 and 9, to maintain the integrity of the experiment result curve, for approximation algorithm CubeTree and MaxMargin, we take the average value of existing approximation ratio as the default value. We present these values with the dotted line in Fig. 8b. These strategies used in this part also be used in subsequent experiments. As can be observed from Fig. 8b, MaxMargin still achieves a better accuracy than CubeTree.

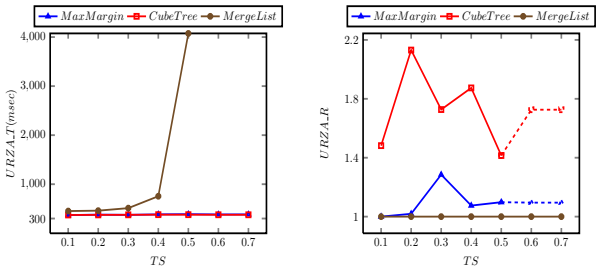
Varying the factor KD. Figs. 9a and 9b show the response time and approximation ratio of three algorithms respectively, when we vary the value of KD. Differing with the method in [24], which fixes the number of keywords associated with objects, in this set of experiments, we set the upper bound for it by KD. For each object, k keywords are generated, where k is a random positive integer less than the upper bound. We vary KD from 3 to 15. It is shown that MaxMargin and CubeTree run much faster than MergeList as the increases of KD. Specifically, we notice that the response time of all three algorithms increases as KD becomes larger. This is because that the number of relevant objects increases as KD increases. Fig. 9b shows that the approximation ratio of MaxMargin changes slightly and is extremely close to 1. In contrast,



(a) (b)

Figure 9: Varying the factor KD

CubeTree performs the worst in terms of accuracy and stability.

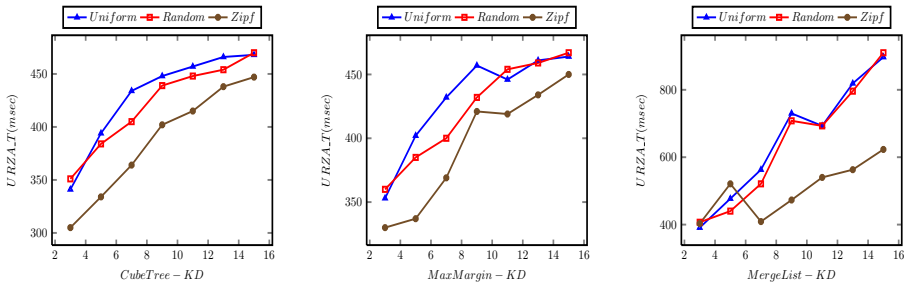


(a) (b)

Figure 10: Varying the factor TS

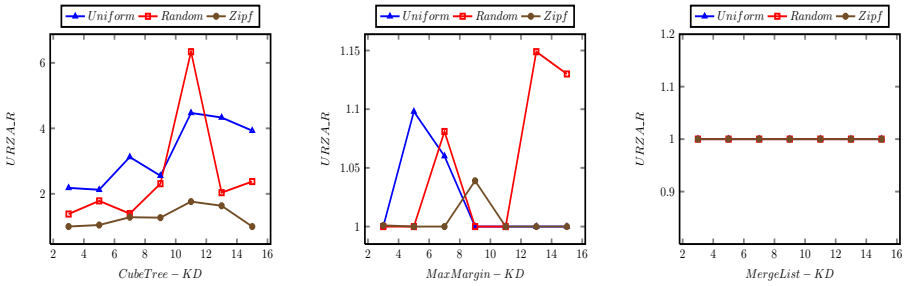
Varying the factor TS. Figs. 10a and 10b show the response time and approximation ratio of three algorithms respectively, when we vary the value of TS. The value of TS determines the size of result set, besides, we can obtain the different combinations of object by adjusting TS. In this set of experiments, we vary TS from 0.1 to 0.7. As can be seen from Fig. 10a that the response time of MergeList increases dramatically. Especially, MergeList runs out of memory when TS takes the value of 0.6. In contrast, CubeTree and MaxMargin are more adaptive to TS and almost not affected by TS as shown in Fig. 10a. This is consistent with earlier findings when we study on the factor of QK. And can be explained with the same reason. Fig. 10b shows again, that MaxMargin performs better than CubeTree in terms of approximation ratio.

The performance on Different Datasets. To further illustrate the performance of our algorithms on different types of datasets. In this set of experiments, we study the performance of our proposed algorithms on different datasets from the aspect of KD. Figs. 11 and 12 show the response time and approximation ratio of our algorithms on different datasets, respectively. Specifically, Figs. 11a, 11b and 11c illustrate the response time of CubeTree, MaxMargin and MergeList respectively. Curves in figures correspond to three datasets, namely, uniform, random and zipf. As can be observed from Fig. 11, that all algorithms are adaptive to zipf dataset. And performance on random dataset is better than on uniform dataset. These phenomenons could be partially attributed to the effect of data skew, which affects the number of relevant objects. Figs. 12a, 12b and 12c illustrate the approximation ratio of CubeTree, MaxMargin and MergeList respectively. And curves in figures correspond to three datasets, namely, uniform, random and zipf. We notice that all algorithms achieve better accuracy on zipf dataset than other two datasets. And uniform dataset performs better than random dataset in terms of accuracy and stability. In a nutshell, all three algorithms perform well on zipf dataset in both response time and approximation ratio. Random dataset



(a) (b) (c)

Figure 11: The Response Time on Different Datasets



(a) (b) (c)

Figure 12: The Approximation Ratio on Different Datasets

with better performance on response time, but worse stability on accuracy than uniform dataset on three algorithms.

Items Of CA	The Scale Of Items
Number of objects (or keywords)	2761823
Number of unique keywords	63
Number of combined objects	20694

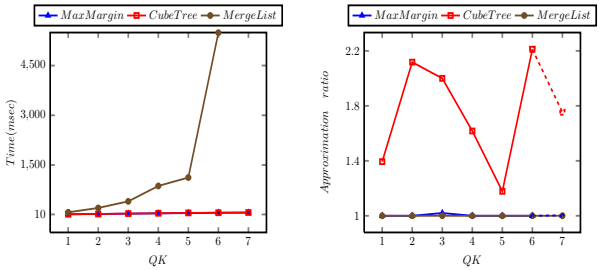
Table 6: The real data of CA

6.3. Results on Real Dataset

In this section, we mainly study the response time and approximation ratio of our proposed algorithms on real data set CA, which was collected from the U.S. Board on Geographic Names(geonames.usgs.gov).

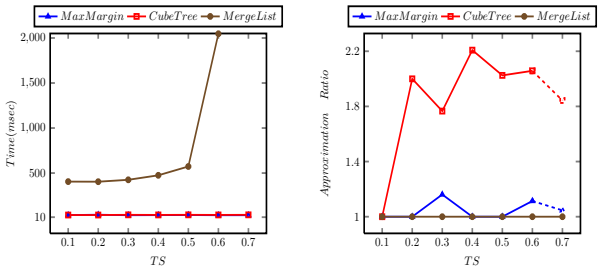
Each object of Geographic Names is a 2D location associated with a set of items describing it (e.g., a geographic name like Locate). We use the *feature class* (e.g., School and Hospital) as the keyword. Since there is only one single feature class associated with each object, which is different from our assumption that each object is associated with several keywords. To address this problem, we associate object o with keywords obtained by aggregating the feature class of all objects whose distance to o within a given threshold. Finally, we obtain 20,694 objects in total as shown in Table 6. And we assign an integer number (e.g., 1,2,3,4,5) randomly for it as the level information, an integer number ranges from 1 to 100 as the cost of o.

Since factors of TK, DS and KD are fixed for CA dataset. We study the effect of QK and TS on real dataset. Figs. 13 and 14 show the experiment results of QK and TS, respectively. As can be observed from Fig. 13a that MergeList runs out of memory when QK takes the value of 7, and both CubeTree and



(a) (b)

Figure 13: Varying the factor QK



(a) (b)

Figure 14: Varying the factor TS

MaxMargin almost not be affected by QK, which is consistent with the observations in Fig. 8a. Fig. 13b shows that the solution of MaxMargin extremely approaches to optimal solution. However, the accuracy of CubeTree is unstable. Fig. 14 shows the performance of our algorithms when we vary TS from 0.1 to 0.7, as shown in Fig. 14, the performance is consistent with the observations in Fig. 10.

From the comprehensive experiments on both synthetic and real-life datasets, we can see that, both CubeTree and MaxMargin run much faster than the exact algorithm MergeList. CubeTree runs slightly faster than MaxMargin, however, MaxMargin with better performance than CubeTree in terms of accuracy and stability of approximation ratio.

7. Conclusions and Future Work

In this paper, we introduce and solve a novel type of queries, namely, KaGWC query. Although CoSKQ can retrieve a group of objects to collectively satisfy the users' needs, however, these exsiting works regardless of the crucial level information of keyword. BKC [11] considers the rating information, but different with ours in query goal. To address this novel problem, we design two approximation algorithms, namely, CubeTree and MaxMargin with provable approximation ratio. Besides, we also propose an exact algorithm MergeList for this problem. Extensive experiments on both real and synthetic datasets were conducted to verify the performance of our proposed algorithms. In a nutshell, both CubeTree and MaxMargin run much faster than the MergeList. CubeTree runs slightly faster than MaxMargin, however, MaxMargin with better performance than CubeTree in terms of accuracy and stability of approximation ratio.

In the future work, there are several interesting research directions. One is to research the KaGWC problem in the road network scenario. Another direction is to take multidimensional level information of keyword into consideration, which can provide the user more appropriate query result. It is also interesting to study other forms of cost function for this problem.

Acknowledgements: This work was supported in part by the National Key Technology Support Program Grants 2012BAH03F02 and 2013BAH62F01, NSFC Grant 61379033, the Fundamental Research Funds for the Central Universities under Grant 2013QNA5020, Zhejiang Province Funds for Science and Technology Innovation Team Grant 2010R50040, and the Key Project of Zhejiang University Excellent Young Teacher Fund (Zijin Plan).

References

[1] X. Cao, L. Chen, G. Cong, J. Guan, N.-T. Phan, and X. Xiao. Kors: Keyword-aware optimal route search system. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1340–1343. IEEE, 2013.

[2] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware optimal route search. *Proceedings of the VLDB Endowment*, 5(11):1136–1147, 2012.

[3] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 373–384. ACM, 2011.

[4] A. Cary, O. Wolfson, and N. Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *Scientific and Statistical Database Management*, pages 87–95. Springer, 2010.

[5] Z. Chen, Y. Liu, R. C.-W. Wong, J. Xiong, G. Mai, and C. Long. Efficient algorithms for optimal location queries in road networks. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 123–134. ACM, 2014.

[6] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

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

[8] G. Cong, H. Lu, B. C. Ooi, D. Zhang, and M. Zhang. Efficient spatial keyword search in trajectory databases. *arXiv preprint arXiv:1205.2880*, 2012.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[10] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *IEEE 24th International Conference on Data Engineering, 2008. (ICDE)*, pages 656–665. IEEE, 2008.

[11] K. Deng, X. Li, J. Lu, and X. Zhou. Best keyword cover search. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2015.

[12] Y. Gao, X. Qin, B. Zheng, and G. Chen. Efficient reverse top-k boolean spatial keyword queries on road networks. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2014.

[13] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *Advances in Spatial and Temporal Databases*, pages 273–290. Springer, 2005.

[14] Z. Li, K. C. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. Ir-tree: An efficient index for geographic document search. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(4):585–599, 2011.

[15] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu. Collective spatial keyword queries: a distance owner-driven approach. In *Proceedings of the 2013 ACM SIGMOD International conference on Management of data*, pages 689–700. ACM, 2013.

[16] J. Lu, Y. Lu, and G. Cong. Reverse spatial and textual k nearest neighbor search. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 349–360. ACM, 2011.

[17] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In *Advances in Spatial and Temporal Databases*, pages 205–222. Springer, 2011.

[18] J. B. Rocha-Junior and K. Nørvåg. Top-k spatial keyword queries on road networks. In *Proceedings of the 15th international conference on extending database technology (EDBT*, pages 168–179. ACM, 2012.

[19] S. Shang, R. Ding, B. Yuan, K. Xie, K. Zheng, and P. Kalnis. User oriented trajectory search for trip recommendation. In *Proceedings of the 15th International Conference on Extending Database Technology (EDBT)*, pages 156–167. ACM, 2012.

[20] Y. Tao and C. Sheng. Fast nearest neighbor search with keywords. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 26(4):878–888, 2014.

[21] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(10):1889–1903, 2012.

[22] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *2011 IEEE 27th International Conference on Data Engineering (ICDE)*, pages 541–552. IEEE, 2011.

[23] B. Yao, M. Tang, and F. Li. Multi-approximate-keyword routing in gis data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 201–210. ACM, 2011.

[24] D. Zhang, Y. M. Chee, A. Mondal, A. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *IEEE 25th International Conference on Data Engineering, 2009. ICDE'09.*, pages 688–699. IEEE, 2009.

[25] D. Zhang, B. C. Ooi, and A. Tung. Locating mapped resources in web 2.0. In *2010 IEEE 26th International Conference on Data Engineering (ICDE)*, pages 521–532. IEEE, 2010.

[26] D. Zhang, K.-L. Tan, and A. K. Tung. Scalable top-k spatial keyword search. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)*, pages 359–370. ACM, 2013.