Geo-Social Keyword Top-k Data Monitoring over Sliding Window

Shunya Nishio(✉), Daichi Amagata, and Takahiro Hara

Department of Multimedia Engineering,
Graduate School of Information Science and Technology,
Osaka University, Yamadaoka 1-5, Suita, Osaka, Japan
nishio.syunya@ist.osaka-u.ac.jp

Abstract. Recently, in many applications, points of interest (PoIs) have generated data objects based on Publish/Subscribe (Pub/Sub) model, and users receive their preferable data objects. Due to the prevalence of location based services and social network services, in addition, locations, keywords, and social relationships are considered to be meaningful for data retrieval. In this paper, we address the problem of monitoring top-k most relevant data objects over a sliding window, w.r.t. distance, keyword, and social relationship. If we have a lot of queries, it is time-consuming to check the result update of all queries. To solve this problem, we propose an algorithm that maintains queries with a Quadtree and accesses only queries with possibilities that a generated data object becomes top-k data. Moreover, we utilize k-skyband technique to quickly update query results. Our experiments using real datasets verify the efficiency of our algorithm.

Keywords: Pub/Sub · Social network · Continuous top-k query

1 Introduction

Due to the prevalence of GPS-enabled mobile devices, data retrieval based on location information and keywords has been becoming essential for many applications which employ Publish/Subscribe (Pub/Sub) model [11,15,17]. That is, users receive only data objects that are relevant to their preferences registered in advance, when points of interest (PoIs) generate data objects [2,4]. In this model, many data objects are continuously generated, so a top-k query that retrieves only k most relevant data objects to users' preferences is useful [2,9]. Moreover, due to the development of social networking service (SNS), such as Facebook and Twitter, data retrieval based on social relationships attracts much attention [1,13]. In particular, by extracting users' preferences with social filtering, it becomes easier to retrieve user requiring data [3]. For example, users have social relationships with their interests on PoIs, such as "like" of Facebook. Figure 1 shows social relationships (described by edges) between u_1, u_2, and PoIs. In this example, u_2 has social relationship with PoI p_1, hence when p_1 generates a data

D. Benslimane et al. (Eds.): DEXA 2017, Part I, LNCS 10438, pp. 409–424, 2017.
DOI: 10.1007/978-3-319-64468-4_31

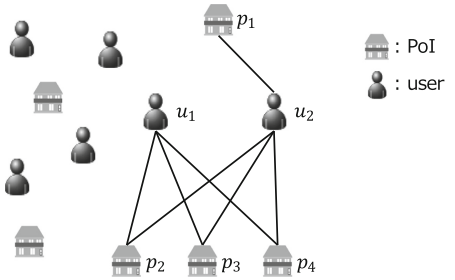


Fig. 1. Social relationships between users and PoIs

object, u_2 receives the data object. It can be seen that the social relationships of u_1 and u_2 are similar, meaning that their preferences would be similar. Although u_1 does not have social relationship with p_1, it is probable that u_1 is interested in p_1. This example implies that Pub/Sub systems should not eliminate chances that u_1 obtains data objects generated by p_1. If u_1 can receive the data objects, u_1 would find new observations, while p_1 can succeed in its advertisements.

In this paper, we consider top-k queries that rank data objects based on distance, keywords, and social relationships. More specifically, we address a problem of top-k data monitoring over a sliding window. Users can specify a query location and keywords and have different social relationships, thus each query has different top-k result. Besides, Pub/Sub systems normally have a huge number of queries, so it is challenging to update the top-k result of each query with real-time. It is also challenging to alleviate top-k re-evaluations when some top-k data objects expire triggered by window sliding.

To solve the above challenges, we propose an algorithm that maintains queries with a Quad-tree [14,16] and accesses only queries with possibilities that a generated data object becomes top-k data. Our Quad-tree supports pruning queries whose top-k results do not change. Furthermore, our algorithm employs k-skyband technique [10] to reduce top-k re-evaluations. Our experiments using real datasets verify the efficiency of our proposed algorithm.

Contribution. We summarize our contributions as follows.

- We address the problem of monitoring top-k data objects based on locations, keywords, and social relationships over a sliding window. To the best of our knowledge, this is the first work which addresses this problem.
- We propose an algorithm that maintains queries with a Quad-tree and accesses only queries with possibilities that a generated data object becomes top-k data. Moreover, we utilize k-skyband technique to quickly update the query result when a top-k data object expires.
- Through experiments using real datasets, we show that our algorithm reduces update time and is more efficient than baseline algorithms.

Organization. In Sect. 2, we define the problem. Section 3 describes our proposed algorithm, and we show our experimental results in Sect. 4. We review related works in Sect. 5. Finally, Sect. 6 concludes the paper.

2 Preliminaries

In this paper, P denotes a set of PoIs, O denotes a set of data objects generated by PoIs, and Q denotes a set of queries. Normally, user are not interested in old data. We thereby avoid providing such data objects. To this end, we employ a sliding window defined below.

Definition 1 (**Sliding window**). *Given a stream of data objects arriving in time order, a sliding window W over the stream with size $|W|$ consists of the most recent $|W|$ data objects.*

Data Object Model. A data object is defined as $o = (id, loc, key, t)$, where $o.id$ is a data identifier (id), $o.loc$ is the location of PoI that has generated o, $o.key$ is a set of keywords, and $o.t$ is the generation time of o.

Query Model. A query issued by a user u is defined as $q_u = (id, loc, key, k)$, where $q_u.id$ is a query id, $q_u.loc$ is a query location, $q_u.key$ is a set of query keywords, and $q_u.k$ is the number of data objects that u is willing to receive.

Our system delivers each data object to its relevant queries. To measure the relevance between a query q_u and a data object o, we define a scoring function. Before introducing the function, we define location score, keyword score, and social score.

Definition 2 (**Location score**). *The location score, $dist(q_u, o)$, between q_u and o is defined as follows.*

$$dist(q_u, o) = 1 - \frac{d(q_u.loc, o.loc)}{MAXloc} \tag{1}$$

where $d(q_u.loc, o.loc)$ is the Euclidean distance between q_u and o, and MAXloc is the maximal possible distance in the space.

Definition 3 (**Keyword score**). *The keyword score, $key(q_u, o)$, between q_u and o is defined as follows.*

$$key(q_u, o) = \frac{2|q_u.key \cap o.key|}{|q_u.key| + |o.key|} \tag{2}$$

Example 2.1. *If $q_u.key = \{w_1, w_2\}$ and $o.key = \{w_2, w_3, w_4\}$, $key(q_u, o) = \frac{2 \times 1}{2+3} = 0.4$.*

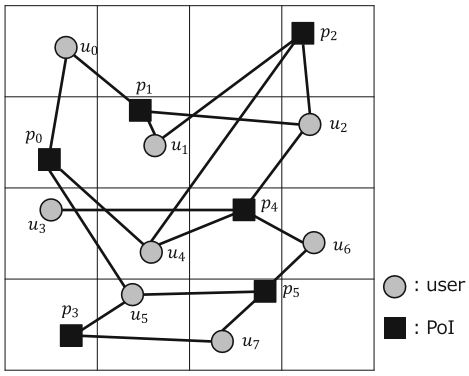


Fig. 2. A social graph

The social score is calculated based on social relationships between users and PoIs. Figure 2 shows a social graph that illustrates social relationships (described by edges) between users and PoIs. Let E denote a set of edges in a social graph, and let $e_{u,p}$ denote the edge between u and p.

Definition 4 (Social score). *The social score, $socio(q_u, p)$, between q_u and p is defined as follows.*

$$socio(q_u, p) = \begin{cases} 1 & (e_{u,p} \in E) \\ \max \frac{2|P_u \cap P_{u'}|}{|P_u| + |P_{u'}|} & (e_{u,p} \notin E) \end{cases} \tag{3}$$

where $P_u = \{\forall p' \in P | \exists e_{u,p'} \in E\}$ and $P_{u'} = \{\forall p' \in P | (\exists e_{u',p} \in E) \wedge (\exists e_{u',p'} \in E)\}$.

Example 2.2. *In Fig. 2, if we assume that p_0 generates o, $socio(q_{u_0}, p_0) = 1$ since $e_{u_0,p_0} \in E$. Next, we assume that p_4 generates o. Since $e_{u_1,p_4} \notin E$, we calculate the similarity of edges between u_1 and u' ($e_{u',p_4} \in E$). For example, $P_{u_1} = \{p_1, p_2\}$ and $P_{u_2} = \{p_1, p_2, p_4\}$, thus the similarity of edges between u_1 and u_2 is computed as $\frac{2 \times 2}{2+3} = 0.8$. Also, the similarities between u_1 and u_3, u_1 and u_4, and u_1 and u_6 are respectively 0, 0.4, and 0. Consequently, $socio(q_{u_1}, p_4) = \max\{0.8, 0, 0.4, 0\} = 0.8$.*

In a real environment, social relationships may be updated (i.e., edge insertions and deletions). It is thus possible to update $socio(q_u, p)$. However, the update frequencies of social relationships between users and PoIs are low, so we do not consider the update of social relationships in this paper, and dealing with such situations is a part of our future work.

Now we are ready to define our scoring function and geo-social keyword top-k queries.

Definition 5 (Scoring function). *The scoring function $s(q_u, o)$ is defined as follows.*

$$s(q_u, o) = dist(q_u, o) + key(q_u, o) + socio(q_u, p) \tag{4}$$

Definition 6 (**Geo-social keyword top-k query**). *Given a set of data objects $O \in W$, a geo-social keyword top-k query q_u retrieves $q_u.k$ data objects with the highest score, and its query result set $q_u.D$ satisfies that $\forall o_i \in q_u.D$, $\forall o_j \in O \setminus q_u.D$, $score(q_u.o_i) \geq score(q_u, o_j)$.*

Problem Statement. Given a massive number of geo-social keyword top-k queries and data objects, we aim to monitor the top-k result of each query with real-time over a sliding window W.

3 Proposed Algorithm

3.1 Overview

First, we describe the overview of the proposed algorithm. We maintain queries with a Quad-tree based on their locations, and then each node stores the query ids, a set of keywords, and the j-th ($j \geq k$) highest scores of the queries in the node. For each leaf node, we store the node id, the query ids in the node, and a digit sequence that shows the path from the root node in a node list (L_N). After construction of Quad-tree, we create a social score list (L^i_{ss}) for each PoI p_i. For each node $n \in L_N$, L^i_{ss} stores the node id and the node social score ss_n, which is the maximum value of social score between $q_u \in n$ and p_i. The construction of Quad-tree and the creation of L_{ss} are preprocessed because queries have been registered in advance.

Given a new data object o, a key challenge is to access only queries such that o becomes their top-k data. First, we create a social score table T_{ss} by combining L_N and L_{ss} of the PoI that has generated o. T_{ss} is used to calculate node social scores. Next, from the root node, we check whether the node is pruned based on the upper bound score. In other words, we prune queries whose top-k results surely do not change. Moreover, we utilize a k-skyband technique to quickly update the query result when a top-k data object expires.

In the following, Sect. 3.2 describes k-skyband. Section 3.3 describes the Quad-tree construction method, and Sect. 3.4 describes the L_{ss} creation method. Finally, Sect. 3.5 describes our algorithm that updates the top-k result of each query.

3.2 *k*-Skyband

We utilize k-skyband to reduce top-k re-evaluations when a top-k data object expires. Followings are formal definitions of dominance and k-skyband.

Definition 7 (**Dominance**). *A data object o_1 is dominated by another data object o_2 w.r.t. a query q_u if both $s(q_u, o_1) \leq s(q_u, o_2)$ and $o_1.t < o_2.t$ hold.*

Definition 8 (k-**skyband**). *The k-skyband of a query q_u contains a set of data objects which are dominated by at most $(k-1)$ other data objects.*

k-skyband contains data objects which may become top-k results when some of the current top-k data objects expire triggered by window sliding. Moreover, the top-k result of q_u belong to the k-skyband of q_u [6]. In our algorithm, k-skyband is updated only when a data object, whose score is higher than the j-th ($j \geq k$) highest score. In the rest of this paper, $q_u.S$ denotes k-skyband of q_u, and $o.dom$ denotes the number of data objects that dominant o.

3.3 Quad-Tree Construction

We adopt the liner Quad-tree structure to maintain a huge number of queries. A Quad-tree is a space partitioning tree structure in which a d-dimensional space is recursively subdivided into 2^d regions. As shown in Fig. 3a, each node resulting from a split is named in the order of NW, NE, SW, and SE. For example, Fig. 3b and c show the space partition and the corresponding tree structure of a simple Quad-tree for a given set of queries $\{q_0, ..., q_3\}$. As shown in Fig. 3c, in this paper, we use a circle and square to denote non-leaf node and leaf node respectively. A leaf node is set *black* if it is not empty, i.e., it contains at least one query. Otherwise, it is a *white* leaf node.

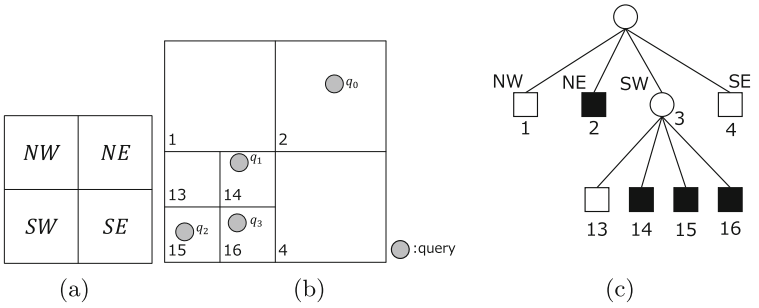
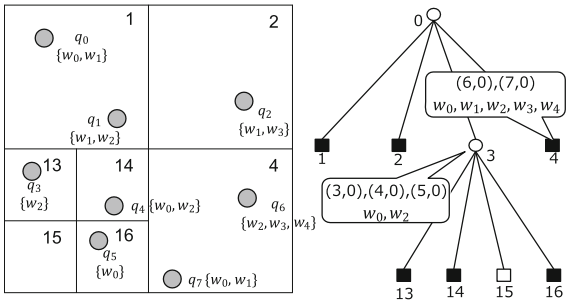


Fig. 3. Illustration of a Quad-tree

Next, we describe the Quad-tree construction method. Each query q_u is assigned into a leaf node based on its location $q_u.loc$. First, the root node is the entire space where all PoIs exist, thus the root node contains all queries. If the number of queries in a node is larger than an integer m, we partition the node into four children nodes and queries are distributed to each node based on their locations, until a node has at most m queries. For example, we assume that users in Fig. 2 issue queries whose locations are their current locations. If $m = 2$, the Quad-tree is constructed as shown in Fig. 4a.

Each node stores the query ids, a set of keywords, and the j-th highest scores of queries in the node. Let $q_u.score_j$ be the j-th highest score of q_u and its initial value is 0. If $q_u.score_j$ is updated, the information stored in nodes that contain q_u is updated. As shown in Fig. 4a, the query id and its $q_u.score_j$ are illustrated by (id, $q_u.score_j$). Node 4 contains q_6 and q_7, thus their ids, the j-th highest



(a)

Node id	Query id	Degit sequence
1	0,1	0
2	2	1
13	3	02
14	4	12
16	5	32
4	6,7	3

(b)

Fig. 4. Construction of a Quad-tree ($m = 2$) and L_N

scores, and a set of their keywords are stored. Node 3 stores the information of q_3, q_4, and q_5 because they are in the subtree rooted at node 3.

Moreover, we create a node list L_N. For each non-empty leaf node, L_N stores the node id, the query ids in the node, and a digit sequence (ds) that shows the path from the root node. ds is calculated from the node id. In a liner Quad-tree, if a node id is x and its parent node id is y, $y = \lfloor \frac{x-1}{4} \rfloor$ and $z = x - 4y$, where z is a remainder. Moreover, z indicates that a node is NW, NE, SE, or SE of its parent node. For example, if $z = 0$, the node is NW of its parent node. By repeating this calculation, the remainder obtained in order is stored as a digit sequence in L_N. For example, since the remainder is obtained in the order of 1 and 2 for node 14, a digit sequence of 12 is stored. As shown in Fig. 4, node 15 does not contain any queries, thus node 15 is not stored in L_N.

3.4 Social Score List

Next, we describe the creation of a social score list (L^i_{ss}) for each PoI p_i. For ease of presentation, we denote a node social score between node n and p as ss_n, and

$$ss_n = \max_{q'_u \in n} socio(q'_u, p). \tag{5}$$

For each node $n \in L_N$, we store the node id and ss_n in L_{ss}. It is however cost-prohibitive that each node stores all ss_n. Thus, if $ss_n = 0$, L_{ss} does not store n.

3.5 Top-k Data Update

Finally, we describe our algorithm that updates the top-k result of each query. We first describe how to process the generated data and then how to process the expired data.

***T_{ss}* Creation.** First, given a new data object, we create a social score table T_{ss} by combining L_N and L_{ss} of the PoI that has generated o. T_{ss} is used to

Algorithm 1. NodeCheck($n, ss_n, o, first, last, d, T_{ss}$)

Input: n: current node, ss_n: node social score of n, o: new incoming data object, $first \cdot last$: a range of use T_{ss}, d: a depth of n, T_{ss}

```
1: Calculate score_ub
2: if score_min ≤ score_ub then
3:    if first = last //n is a leaf-node then
4:       for ∀q_u ∈ n do
5:          Update(q_u, o, T_ss[first].ds)
6:    else
7:       for ∀n_c ∈ [NW, NE, SW, SE] do
8:          count_{n_c} ← 0
9:       for i = first to last do
10:         if d-th digit of T_ss[i].ds = 0 then
11:            x ← NW
12:         else if d-th digit of T_ss[i].ds = 1 then
13:            x ← NE
14:         else if d-th digit of T_ss[i].ds = 2 then
15:            x ← SW
16:         else
17:            x ← SE
18:         count_x ← count_x + 1
19:         if ss_x < T_ss[i].ss_n then
20:            ss_x ← T_ss[i].ss_n
21:      d ← d + 1
22:      for ∀n_c ∈ [NW, NE, SW, SE] //in order of NW, NE, SW, and SE do
23:         if count_{n_c} > 0 then
24:            last ← first + count_{n_c} − 1
25:            NodeCheck(n_c, ss_{n_c}, o, first, last, d, T_ss)
26:            first ← first + count_{n_c}
27: else
28:    for ∀q_u ∈ n do
29:       if q_u.s_p ≤ score_ub then
30:          q_u.s_p ← score_ub
```

calculate the node social score. T_{ss} stores the node ids, the node social scores (ss_n), and the digit sequences (ds) of all nodes in L_N. That is, each entry of T_{ss} is $<n, ss_n, ds>$. For each node n, if $n \in L_{ss}$, T_{ss} stores $ss_n \in L_{ss}$. Otherwise, T_{ss} stores 0.

Node Check. After T_{ss} creation, from the root node, we check nodes and access only queries with possibilities that the score of o is not less than their j-th highest scores. Note that ss_n of the root node is 1. Algorithm 1 shows our algorithm. For each node n, we compare $score_{min}$ and the upper bound score ($score_{ub}$), where $score_{min}$ is the minimum value of $q_u.score_j$ ($q_u \in n$) and $score_{ub}$ of o is

$$score_{ub} = dist(n, o) + key(n, o) + ss_n. \tag{6}$$

Note that $dist(n, o)$ is calculated by using the minimum distance from o to n. If $o \in n$, however, $dist(n, o) = 1$. $key(n, o)$ is calculated as follows:

$$key(n, o) = \frac{2|n.key \cap o.key|}{|n.key \cap o.key| + |o.key|}, \tag{7}$$

where $n.key$ is a set of keywords stored in n. For each query $q_u \in n$, $dist(n, o) \geq dist(q_u, o)$ and $key(n, o) \geq key(q_u, o)$ hold. The top-k result of q_u is updated, if $score(q_u, o) \leq q_u.score_k$. Then the following theorem holds.

Theorem 1. *Given a node n, if $score_{min} > score_{ub}$, we can safely prune all the queries in n.*

Proof. The top-k result of $q_u \in n$ is updated, if $score(q_u, o) \leq q_u.score_k$. If $score_{min} > score_{ub}$, $score_{min} > dist(n, o) + key(n, o) + ss_n$. Moreover $q_u.score_k \geq q_u.score_j \geq score_{min}$ and $dist(n, o) + key(n, o) + ss_n \geq dist(q_u, o) + key(q_u, o) + socio(q_u, p) = s(q_u, o)$. Thus if $score_{min} > score_{ub}$, $q_u.score_k > s(q_u, o)$. Theorem 1 therefore holds. □

Next, we compare $score_{ub}$ and $score_{min}$ of the current node (Line 2). If $score_{ub} < score_{min}$, we prune all the queries in the current node because their k-skyband is not updated. At that time, for each query q_u, if $q_u.s_p < score_{ub}$, we assign $score_{ub}$ to $q_u.s_p$, where $q_u.s_p$ is the maximum value of $score_{ub}$ when q_u is pruned. (We describe the detail of $q_u.s_p$ later.) If $score_{ub} \geq score_{min}$, we iteratively check the children nodes of the current node. For each child node, we count the number of leaf nodes in T_{ss} and the subtree rooted at the child node. We simultaneously calculate ss_n of the children nodes (NW, NE, SW, and SE) (Lines 6–21). Next, for each child node that contains at least one leaf node, we similarly check nodes in order of NW, NE, SW, and SE (Lines 22–26). If $score_{ub} \geq score_{min}$ at a leaf node, for each query in the node, we update its k-skyband. That is, we execute Algorithm 2 that updates k-skyband, top-k result, and the information stored in Quad-tree (Lines 3–5). (We describe the detail of the algorithm later.)

Example 3.1. *In Fig. 4, we assume that $q_6.score_j = 1.8$ and $q_7.score_j = 2.2$. We furthermore assume that p_0 in Fig. 2 generates a data object o where $o.key = \{w_1, w_2, w_5\}$. We then check node 4 in Fig. 4. Since $n.key = \{w_0, w_1, w_2, w_3, w_4\}$ and $o.key = \{w_1, w_2, w_5\}$, $n.key \cap o.key = \{w_1, w_2\}$. Thus, $key(n, o) = \frac{2 \times 2}{2+3} = 0.8$. Now, we assume that $dist(n, o) = 0.6$ and $ss_n = 0.8$. In this situation, $score_{ub} = 0.6 + 0.8 + 0.8 = 2.2$. Since $score_{ub}(2.2) > score_{min}(1.8)$, we access queries in node 4 and update their k-skyband and top-k results.*

***k*-Skyband and Top-k Result Update.** Algorithm 2 shows our update algorithm. If $score_{ub} \geq score_{min}$ at a leaf node, for each query in the node, we update k-skyband, top-k result, and the information stored in Quad-tree. First, we calculate $score(q_u.o)$ and update $o'.dom$ ($\forall o' \in q_u.S$) (Lines 1–4). If $o'.dom \geq k$, we delete o' from $q_u.S$. Next, if $score(q_u, o) \geq q_u.score_k$, we update the top-k result $q_u.D$ from $q_u.S$ (Lines 8–9). In our algorithm, the information stored in Quad-tree is updated triggered by k-skyband update. For each node that contains q_u whose k-skyband is updated, we update $q_u.score_j$ (Lines 10–12).

Algorithm 2. Update(q_u, o_{in}, ds)

Input: q_u: a query, o_{in}: a new incoming data, ds: a digit sequence

```
1: Calculate s(q_u, o_in)
2: for ∀o ∈ q_u.S do
3:    if s(q_u, o_in) ≥ s(q_u, o) then
4:       o.dom ← o.dom + 1
5:       if o.dom ≥ q_u.k then
6:          q_u.S ← q_u.S − {o}
7: q_u.S ← q_u.S + {o_in}
8: if s(q_u, o_in) ≥ q_u.score_k then
9:    Update q_u.D from q_u.S
10: N ← A set of nodes that are calculated by ds
11: for ∀n ∈ N do
12:    Update q_u.score_j
```

Algorithm 3. Topk-Reevaluation(o)

Input: o: an expired data object

```
1: for each query q_u such that o ∈ q_u.S do
2:    q_u.S ← q_u.S − {o}
3:    if o ∈ q_u.D then
4:       if |q_u.S| ≥ q_u.k then
5:          Extract q_u.D from q_u.S
6:          if q_u.s_p ≥ q_u.score_k then
7:             Extract q_u.D, q_u.S from W
8:       else
9:          Extract q_u.D, q_u.S from W
```

Data Expiration. Algorithm 3 shows our top-k re-evaluation algorithm. First, for each query whose k-skyband contains the expired data object o, we delete o from $q_u.S$. We then update the top-k result of query q_u where $o \in q_u.D$. The k data objects with the highest score in $q_u.S$ become top-k result. If $|q_u.S| < k$ or a data object with score less than $q_u.s_p$ becomes top-k result, we need to re-evaluate its top-k result from W. Recall that $q_u.s_p$ is the maximum value of $score_{ub}$ when q_u is pruned. Thus, if a data object with score less than $q_u.s_p$ becomes top-k result, it is possible that a data object pruned before becomes top-k result. We compute $score(q_u, o')$ of a data object $o' \in W$, and then update $q_u.D$ and $q_u.S$.

4 Experiments

In this section, we provide an experimental evaluation on the performance of our proposed algorithm. Since this is the first work addressing the problem of monitoring top-k data objects based on locations, keywords, and social relationships over a sliding window, we compare our algorithm with following three baseline algorithms.

- **baseline 1.** This algorithm does not utilize our query index. Given a new data object, we update k-skyband and top-k results of all queries. When a top-k data object expires, we update the top-k result from k-skyband.
- **baseline 2.** This algorithm utilizes our query index, but does not utilize the k-skyband technique. Given a new data object, we access only queries with possibilities that the score of the data object is not less than the j-th highest score of their queries. When a top-k data object expires, we check all data objects in W to update the top-k result.
- **baseline 3.** This algorithm utilizes our query index, but does not utilize the k-skyband technique. The difference with baseline 2 is that, when we calculate the score of a data object o, we utilize L_{ss}. Specifically, when we calculate the social score of o, if the leaf node n containing q_u is in L_{ss} of the PoI that has generated o, we assign $ss_n \in L_{ss}$ to the social score. Otherwise, we assign 0 to the social score. We then calculate the approximate score of o and compare the score with $q_u.score_k$. If the score is higher than $q_u.score_k$, we calculate the accurate score of o and update the top-k result.

4.1 Experimental Settings

All algorithms were implemented in C++, and all experiments were conducted on a PC with 3.47 GHz Intel Xeon processor and 192 GB RAM.

Dataset. Our experiments were conducted on two real datasets: Yelp[1] and Brightkite[2]. The statistics of the two datasets are summarized in Table 1. Each dataset contains PoIs with location and check-ins, thus we utilize the check-ins as social relationships between users and PoIs. For $o.key$, we randomly select a keyword set from Yelp reviews. Each keyword set contains 1 to 11 keywords. Figure 5 shows the histogram of the number of keywords in a keyword set.

Table 1. Dataset statistics

	Yelp	Brightkite
Number of users	366,715	50,687
Number of PoIs	60,785	772,631
Number of check-ins	1,521,160	1,072,965

Parameters. Table 2 summarizes the parameters and the default values (bold ones) used in the experiments. We used 10 and $2k$ as default values of m and j. We generated a top-k query for each user. For $q_u.loc$, we assign a random location in the space where PoIs exist. We randomly selected a keyword set from Yelp reviews as $q_u.key$ and $q_u.k$ was a random number between 1 to k_{max}.

[1] http://www.yelp.com/dataset_challenge/.
[2] http://snap.stanford.edu/data/loc-brightkite.html.

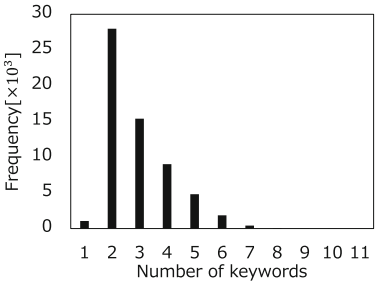


Fig. 5. Histogram of the number of keywords in a keyword set

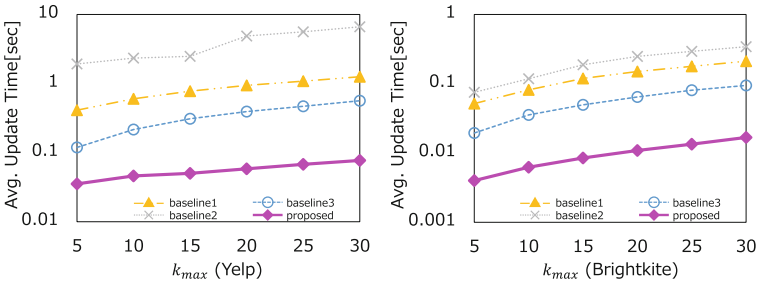
Table 2. Configuration of parameters

Parameter	Value
Maximum value of $q_u.k$, k_{max}	5, **10**, 15, 20, 25, 30
$\|Q\|$ [$\times 10^3$]	**50**, 100, 150, 200, 250, 300, 350 (Yelp)
	10, 20, 30, 40, 50 (Brightkite)
$\|W\|$ [$\times 10^6$]	**0.5**, 1, 1.5, 2, 2.5, 3

4.2 Experimental Result

We report the average update time (the average time to finish updating the top-k result of each query triggered by window sliding [sec]) of each algorithm.

Effect of k_{max}. We examine the impact of k_{max}, and Fig. 6 shows the result. As shown in Fig. 6, our algorithm updates the top-k results faster than the baseline algorithms. All the algorithms need longer update time as k_{max} increases. The difference in the update time between baselines 2 and 3 in Yelp case is larger than that in Brightkite case. In Yelp dataset, it can be seen that the number of POIs is smaller than the number of users, meaning that users would check in the

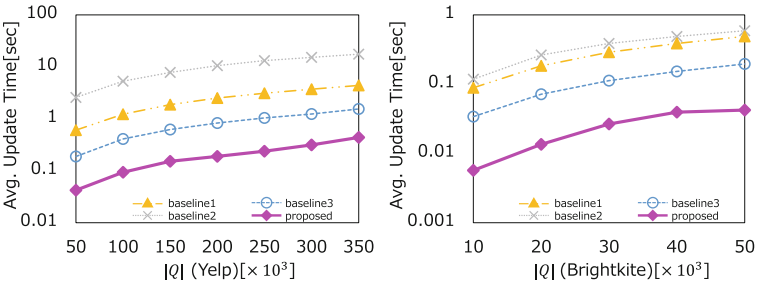


(a) Yelp (b) Brightkite

Fig. 6. Effect of k_{max}

same PoIs. It thus takes much time to calculate social score, and the performance of baseline 2 that calculates social scores of all data objects decreases.

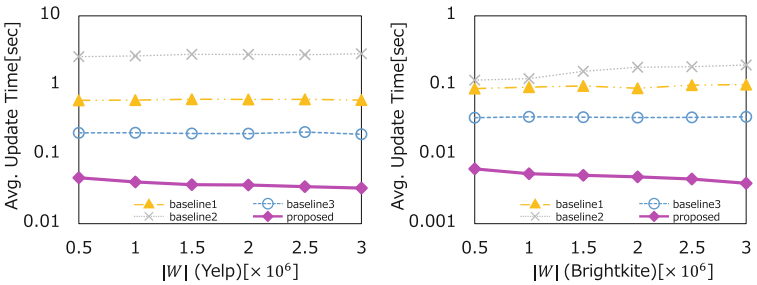
Effect of $|Q|$. We next examine the impact of $|Q|$, and Fig. 7 shows the result. As shown in Fig. 7, all the algorithms need longer update time as $|Q|$ increases and our algorithm updates the top-k results faster than the baseline algorithms. Specifically, the update time of our algorithm is at least four times faster than those of the baseline algorithms.



(a) Yelp (b) Brightkite

Fig. 7. Effect of $|Q|$

Effect of $|W|$. We finally examine the impact of $|W|$, and Fig. 8 shows the result. When $|W|$ increases, the update time of each baseline algorithm is constant or increases. However, the update time of our algorithm decreases. The update time includes generated data processing time (i.e., GDP) and expired data processing time (i.e., EDP). When we increase $|W|$, the GDP time of baselines 2 and 3, and our algorithm decreases, due to the fact that a large window size usually leads to top-k results with higher score. Thus, a new data object affects less queries, resulting in shorter GDP time. On the other hand, the GDP time of baseline



(a) Yelp (b) Brightkite

Fig. 8. Effect of $|W|$

1 is constant because baseline 1 always checks all queries. With increasing $|W|$, the possibility that an expired data object is contained in the top-k results of queries becomes low. The number of top-k re-evaluations hence decreases. On the other hand, when a top-k data object expires, baselines 2 and 3 check all data objects in W. When $|W|$ increases, the EDP time of baselines 2 and 3 therefore increases. Recall that, we utilize k-skyband technique in baseline 1 and our algorithm, thus we can quickly update the top-k result when a top-k data object expires.

5 Related Work

In this section, we focus on the existing geo-social keyword search and Pub/Sub systems and review them.

5.1 Geo-Social Keyword Search

Several works addressed the problem of geo-social keyword search [1,13]. The work in [13] introduced the Social-aware top-k Spatial Keyword (SkSK) query and proposed Social Network-aware IR-Tree (SNIR-tree) for the processing of SkSK queries. SNIR-tree is an R-Tree, where each node also contains a set of users relevant to the POIs indexed by the subtree rooted at the node. In this paper, however, the definition of social relationships is different from ours and this tree structure does not support efficient top-k data monitoring. Also, the work in [1] considers snapshot queries over static data while our problem focuses on continuous query over streaming data.

5.2 Pub/Sub System

Numerous works studied Pub/Sub systems. Nevertheless, most of the existing Pub/Sub systems, e.g., [7,8,12,15], do not consider spatial information. Recently, spatial-keyword Pub/Sub system has been studied in a line of work (e.g., [2,4,5,10,11]). Among them, the works in [5,11] study the boolean matching problem while the work in [4] studies the similarity search problem, where each query has a pre-given threshold. However, these works are different from ours because they do not consider top-k data monitoring. The only works in [2,10] have addressed the problem of spatio-textual top-k publish/subscribe. The work in [10] has proposed a novel system, called Skype (Top-k Spatial-Keyword Publish/Subscribe), that monitors top-k data objects based on locations and keywords over a sliding window. In this paper, however, our data retrieval is based on social relationships and we need to store the information of each PoI because each PoI has different social relationships. It thus is hard to apply the approach to our addressing problem. We similarly cannot apply the approach proposed in [2] because the work does not consider social relationships and a sliding window model.

6 Conclusion

In this paper, we addressed the problem of monitoring top-k most relevant data objects, w.r.t. distance, keyword, and social relationship, with sliding window setting. To quickly update the top-k result of each query, we proposed an algorithm that maintains queries with a Quad-tree and accesses only queries with possibilities that a generated data object becomes the top-k data. Moreover, we utilize k-skyband technique to quickly update the query result when a top-k data object expires. We evaluated our proposed algorithm by experiments on real data, and the results show that our algorithm reduces the update time and is more efficient than the three baseline algorithms.

In this paper, we did not consider the update of social relationships. As part of our future work, we plan to design an algorithm which can deal with updates of social relationships.

Acknowledgement. This research is partially supported by the Grant-in-Aid for Scientific Research (A)(26240013) of the Ministry of Education, Culture, Sports, Science and Technology, Japan, and JST, Strategic International Collaborative Research Program, SICORP.

References

1. Ahuja, R., Armenatzoglou, N., Papadias, D., Fakas, G.J.: Geo-social keyword search. In: Claramunt, C., Schneider, M., Wong, R.C.-W., Xiong, L., Loh, W.-K., Shahabi, C., Li, K.-J. (eds.) SSTD 2015. LNCS, vol. 9239, pp. 431–450. Springer, Cham (2015). doi:10.1007/978-3-319-22363-6_23
2. Chen, L., Cong, G., Cao, X., Tan, K.-L.: Temporal spatial-keyword top-k publish/subscribe. In: ICDE, pp. 255–266 (2015)
3. Groh, G., Ehmig, C.: Recommendations in taste related domains: collaborative filtering vs. social filtering. In: SIGGROUP, pp. 127–136 (2007)
4. Hu, H., Liu, Y., Li, G., Feng, J., Tan, K.-L.: A location-aware publish/subscribe framework for parameterized spatio-textual subscriptions. In: ICDE, pp. 711–722 (2015)
5. Li, G., Wang, Y., Wang, T., Feng, J.: Location-aware publish/subscribe. In: KDD, pp. 802–810 (2013)
6. Mouratidis, K., Bakiras, S., Papadias, D.: Continuous monitoring of top-k queries over sliding windows. In: SIGMOD, pp. 635–646 (2006)
7. Mouratidis, K., Pang, H.: Efficient evaluation of continuous text search queries. IEEE TKDE **23**(10), 1469–1482 (2011)
8. Sadoghi, M., Jacobsen, H.-A.: BE-Tree: an index structure to efficiently match Boolean expressions over high-dimensional discrete space. In: SIGMOD, pp. 637–648 (2011)
9. Shraer, A., Gurevich, M., Fontoura, M., Josifovski, V.: Top-k publish-subscribe for social annotation of news. PVLDB **6**(6), 385–396 (2013)
10. Wang, X., Zhang, Y., Zhang, W., Lin, X., Huang, Z.: Skype: top-k spatial-keyword publish/subscribe over sliding window. PVLDB **9**(7), 588–599 (2016)
11. Wang, X., Zhang, Y., Zhang, W., Lin, X., Wang, W.: AP-Tree: efficiently support continuous spatial-keyword queries over stream. In: ICDE, pp. 1107–1118 (2015)

12. Whang, S.E., Garcia-Molina, H., Brower, C., Shanmugasundaram, J., Vassilvitskii, S., Vee, E., Yerneni, R.: Indexing Boolean expressions. PVLDB **2**(1), 37–48 (2009)
13. Wu, D., Li, Y., Choi, B., Xu, J.: Social-aware top-k spatial keyword search. In: MDM, pp. 235–244 (2014)
14. Zhang, C., Zhang, Y., Zhang, W., Lin, X.: Inverted linear quadtree: efficient top k spatial keyword search. In: ICDE, pp. 901–912 (2013)
15. Zhang, D., Chan, C.-Y., Tan, K.-L.: An efficient publish/subscribe index for e-commerce databases. PVLDB **7**(8), 613–624 (2014)
16. Zhang, D., Tan, K.-L., Tung, A.K.: Scalable top-k spatial keyword search. In: EDBT, pp. 359–370 (2013)
17. Zheng, K., Su, H., Zheng, B., Shang, S., Xu, J., Liu, J., Zhou, X.: Interactive top-k spatial keyword queries. In: ICDE, pp. 423–434 (2015)