Continuous Top-k Spatial Keyword Search on Dynamic Objects

Yuyang Dong
NEC Corporation, Japan
y-dong@aj.jp.nec.com

Hanxiong Chen
University of Tsukuba, Japan
chx@cs.tsukuba.ac.jp

Jeffrey Xu Yu
The Chinese University of Hong Kong, China
yu@se.cuhk.edu.hk

Hiroyuki Kitagawa
University of Tsukuba, Japan
kitagawa@cs.tsukuba.ac.jp

ABSTRACT

As the popularity of SNS and GPS-equipped mobile devices rapidly grows, numerous location-based applications have emerged. A common scenario is that a large number of users frequently change location and interests (represented by a set of keywords) in real time; e.g., a user watches news, videos, and blogs while moving outside. Many online services have been developed based on continuously searching and monitoring spatial-keyword objects. For instance, a real-time coupon delivery system searches for potential customers by matching their locations and interested keywords, and then sends coupons to attract potential customers.

In this paper, we investigate the case of *dynamic* spatial-keyword objects whose locations and keywords change over time. We study the problem of continuously searching for top-k dynamic spatial-keyword objects for a given set of queries. Answering this type of queries benefit many location-aware recommender systems such as real-time E-coupon delivery. We develop a solution that can be applied on a grid or a quadtree index. To deal with the changing locations and keywords of objects, our solution first finds the set of queries whose results are affected by the change and then updates the results of these queries. For efficient update, we maintain a small set of cells to reduce the computation cost of top-k reevaluation. We analyze the computation cost of our method and discuss how to tune for the best number of cells. We also discuss batch processing to cope with the case when a large number of objects change locations and keywords at the same time. Experimental results on real datasets demonstrate the efficiency of our method and its superiority over alternative solutions.

PVLDB Reference Format:
Yuyang Dong, Hanxiong Chen, Jeffery Xu Yu, and Hiroyuki Kitagawa. Continuous Top-k Spatial Keyword Search on Dynamic Objects. *PVLDB*, 13(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

Proceedings of the VLDB Endowment, Vol. 13, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

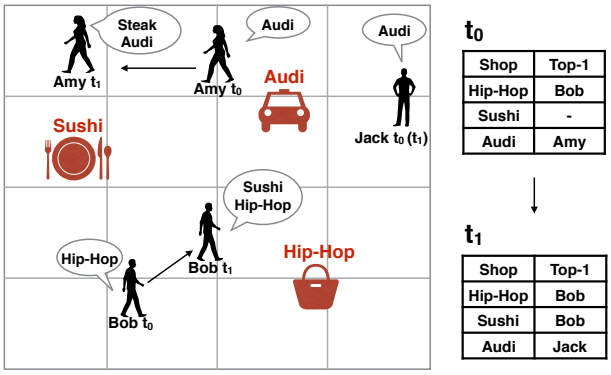


Figure 1: E-coupon recommender system.

1. INTRODUCTION

Nowadays many people prefer to access the Web using mobile devices. The prevalence of GPS-equipped mobile devices generates massive volume of data that contain both spatial and textual information. At the same time, many location-aware service have been developed by managing spatial-keyword data. E.g., a user may look for the information she's interested in using her mobile phone when moving outside, while a location-aware recommender system may discover potential customers for stores by continuously searching for nearby people based on the interests they would like to share, usually represented by a set of keywords.

Despite various types of queries on spatial-keyword data being studied in the last decade, existing studies focus on dealing with static objects or moving objects that only changes locations. Yet, real-world objects are often dynamic and change both locations and keywords over time. In this paper, we explore this scenario and study the problem of continuously searching for top-k dynamic spatial-keyword objects: given a set of dynamic spatial-keyword objects whose locations and keywords may vary from time to time, and a set of (static) queries represented by locations and keywords, our task is to monitor the top-k results for each query, ranking by a scoring function that takes a linear combination of spatial and keyword similarities. This problem has many underlying applications, e.g., location-aware recommender systems and spatial-keyword data analytics. Next we show how the solution to this problem benefits real-world applications with a motivating example.

Figure 1 shows a scenario that explains how this type of

queries works on real applications. Consider an E-coupon recommender system, where a hip-hop cloth store, a sushi restaurant and an Audi dealer are registered. Three registered users are using mobile phones. Suppose we search for top-1 results; i.e, the system monitors the top-1 users w.r.t. the three stores and sends out coupons. At time t_0, Bob is watching a hip-hop music video, Amy is searching for Audi, and Jack is reading news about Audi. Bob becomes the top-1 result of the hip-hop store, as he is the only user that matches keyword hip-hop. Both Amy and Jack are associated with keyword Audi, but Amy is closer to the Audi dealer and becomes the top-1 result. Nobody is associated with keyword sushi. So the top-1 result for the sushi restaurant is empty. At time t_1, Bob is moving northeast and starts to watch an eating show about sushi. So his keywords become hip-hop and sushi [1]. Amy moves away from the Audi dealer and searches "steak near me". So her keywords become Audi and steak. Jack is staying and still reading news about Audi. The system keeps Bob at the top-1 result of the hip-hop store, and recognizes him as the new top-1 of the sushi restaurant. As Jack becomes closer than Amy to the Audi dealer, he replaces Amy as the top-1 of the Audi dealer.

To the best of our knowledge, this is the first work targeting the case when objects change both spatial and textual attributes. Although one may regard the update of an object as a deletion followed by an insertion and convert dynamic objects to static ones, it does not mean that existing methods for related problems – e.g., CIQ [3] and SKYPE [15] for location-aware publish/subscribe systems – can be simply applied to this case. CIQ only considers objects (referred to as messages in [3, 15]) appended to the system without deletions and thus is not applicable. SKYPE deletes an object only when it expires from the sliding window. Since the object update is ad-hoc in our problem, the window size has to be infinitely small to capture the semantics of a deletion-insertion pair, rendering the top-k reevaluation module in SKYPE not work.

1.1 Challenges and Contributions

The first challenge originates from the large number of queries. It is prohibitive to check every query and recompute the top-k results. On the other hand, the number of affected queries (i.e., the queries that change their top-k results) is usually very small once an update occurs. This challenge is akin to the case in location-aware publish/subscribe systems. Both CIQ and SKYPE utilize an inverted index for keywords and a quadtree for spatial information, in order to quickly identify the queries (referred to as subscriptions in [3, 15]) affected by an incoming object. SKYPE further adopts the prefix filtering technique used in set similarity search [1, 19] to limit the inverted index access to only keywords with high weights. However, using an inverted index to look for affected queries is not an efficient option in our problem. This is because it is common that an object in our problem has only frequent keywords, and an update can be an insertion of a frequent keyword to an object. This results in access to long postings lists in the inverted index and hence considerable overhead.

To address the first challenge, we propose the notion of *influential circles*. By bounding the keyword similarity of an object to a query, each query has its influential circle that demarcates the region in which an object is possible to become a top-k result of the query. Queries are indexed in the cells (of a grid or a quadtree) that overlap their influential circles. Thus, whenever an object changes its status, we only need to check the new cell of the object to quickly find the affected queries. In addition, despite the inefficiency for our problem, a subset of the pruning techniques of SKYPE are orthogonal to our method, and can be adapted and integrated into our method to further improve the efficiency. Such possibilities are discussed.

The second challenge is the top-k reevaluation, i.e., to compute the top-k results of the affected queries. Since the update of an object may cause the object to get out of the top-k list of some queries (e.g., when an object moves away from a query), it is time-consuming to refill the top-k of these queries from scratch. Most related studies adopt a buffering strategy to store a list of non-top-k objects for each query. Then the buffered objects are used when a reevaluation is needed. E.g., kmax [21] maintains top-k' results where k' is a value between k and a maximum buffer size $k_{\max}$; SKYPE uses a cost-based k-skyband buffer by setting a threshold of score. Such object-based buffering is inefficient for our problem because a reordering of the objects in the buffer is required whenever an update occurs.

We adopt a *cell-based buffering* strategy to tackle the second challenge. The rationale behind is that even though the update of objects may be frequent, the maximum/minimum score of the objects in a cell does not change frequently. We first propose a method that maintains all the cells for each query, and then improve it by keeping only a subset of cells that contain the objects needed for top-k reevaluation. By the improvement, we achieve a small number of cells maintained for each query and further reduce the frequency of updates to the buffer.

In addition to the above proposed techniques, we theoretically analyze the computation cost of our method, so as to find the best number of cells for query processing. We also discuss batch processing using our method, in case a large number of objects change status at the same time. We conduct experiments on two real datasets and a synthetic dataset. The results demonstrate that the proposed techniques of our method are effective in reducing query processing cost and contributing to the overall speedup over alternative solutions.

Our contributions are summarized as follows.

- We study a new type of queries to continuously search top-k results for dynamic spatial-keyword objects that may change locations and keywords over time.
- We propose a query processing method that comprises an affected query finder and a top-k refiller to address the technical challenges of the studied problem.
- For efficiency, we theoretically analyze the cost of our method to tune the number of cells in the index. We also extend our method to handle the case of batch processing.
- We conduct extensive experiments on real datasets. The results demonstrate the effectiveness of the components of our method and the superiority of our method over alternative solutions.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces basic concepts and defines the problem. Section 4 provides the overview of our solution and briefly introduces our index. Sections 5 and 6 present the algorithms of the two main components of our method: affect query finder and top-k refiller, respectively. We analyze the computation cost of our method for cell size tuning in Section 7. Experimental results are reported and analyzed in Section 9. Section 10 concludes the paper.

2. RELATED WORK

[1] In this example, previous keywords do not immediately disappear. The detailed setting depends on the application.

Table 1: Comparison to related studies.

Research	Objects	Queries	Attributes
This work	dynamic	static	spatial-keyword
[3, 15]	streaming	streaming	spatial-keyword
[8, 9, 17, 18, 23]	static	moving	spatial-keyword
[4, 5, 11]	static	static	spatial-keyword
[12, 13, 20, 22]	moving	static	spatial

Due to the space limitation, we review the most related studies here.

Location-aware publish/subscribe systems. The studies most related to our work focus on dealing with spatial-keyword objects in a publish/subscribe system. Users register their interests as continuous queries into the system, and then new streaming objects are delivered to relevant users. The case of boolean matching was studied in [2, 10, 16], while the case of scoring function combining spatial and keyword similarities were considered in [3, 15]. The major difference of this line of work from ours is that they do not consider dynamic objects. Moreover, as we mentioned in Section 1, the techniques in this line of work cannot deal with ad-hoc deletion/insertion pairs (as the update operation in our problem can be regarded as a deletion followed by an insertion of objects).

Moving queries on static objects. While we focus on dynamic objects and static queries, a body of work studied the case of moving queries on static objects. They target applications like searching top-k gas-stations for a moving car. Wu *et al.* [17] proposed to answer continuously moving top-k spatial-keyword (MkSK) queries using safe-regions on multiplicatively weighted Voronoi cells. Huang *et al.* [9] studied MkSK queries with a general weighted sum ranking function and proposed to use hyperbola-based safe-regions to filter objects. Zheng *et al.* [23] studied continuous boolean top-k spatial-keyword queries in a road network. Guo *et al.* [8] studied continuous top-k spatial-keyword queries with a combined ranking function.

Snapshot spatial-keyword search. Searching static geo-textual objects for spatial-keyword queries have been extensively studied, e.g., for boolean matching [6] or using a scoring function [14]. We refer readers to [4, 5, 11] for various problem settings and methods. The studies in this category focus on a snapshot query on static datasets, whereas our problem is based on a continuous query with dynamic spatial-keyword objects.

Monitoring moving objects. Another line of work [12, 13, 20, 22] aims at keeping the kNN moving objects w.r.t. a fixed query point. Mouratidis et al. [12] proposed the notion of influence region in a grid index. These solutions only consider the spatial similarity and thus cannot be directly used for our problem. E.g., the objects outside the influence region may be very high in keyword similarity and thus ranked higher than those inside the influence region.

Table 1 compares our work to existing studies. It is noteworthy to mention that our work is the only one that considers dynamic objects that may change both locations and keywords. Streaming objects/queries appear and disappear in a sliding window but do not change attributes. Moving objects/queries may change locations.

3. PRELIMINARIES

Table 2: Frequently used notations.

Symbol	Description
o, O	an object, a set of objects
q, Q	a query, a set of queries
$top\text{-}k(q,t)$	the top-k objects of q at time t
$kScore(q,t)$	the score of the k-th object in $top\text{-}k(q,t)$
o, o'	the old/new status of an object
IC_q	the influential circle of q
c	a cell in a grid index
n, n^2	number of rows/columns, number of cells in a grid
$o.c$	the cell in which o is located
Q_o	the set of queries to which o is a top-k object
$Q_{o'}$	the set of queries s.t. $SimST(o',q) > kScore(q,t)$
$q.o_{k+1}$	the $(k+1)$-th result of q at time t
$maxscore(c,q)$	the maximum score of the objects in c w.r.t. q
$minscore(c,q)$	the minimum score of the objects in c w.r.t. q
$q.CL$	a sorted cell list of q
$q.PCL$	a partial sorted cell list of q

Table 2 lists the notations frequently used in this paper.

Definition 1 (Dynamic Spatial-Keyword Object) *A dynamic spatial-keyword object o is defined as $(o.\rho, o.\psi)$. $o.\rho$ is the up-to-date location of o, represented by spatial coordinates. $o.\psi$ keeps track of the up-to-date keywords of o, represented by a set. Both $o.\rho$ and $o.\psi$ are dynamic and change over time.*

Definition 2 (Spatial-Keyword Query) *A spatial-keyword query q is defined as $(q.\rho, q.\psi, q.\alpha)$, where $q.\rho$ is a location, $q.\psi$ is a set of keywords, and $q.\alpha$ is a query-specified parameter to balance spatial and keyword similarities. The three attributes are all static.*

Example 1 *Consider the example In Figure 1. We regard persons as objects and shops as queries. At time t_0, we have three objects: $(o_1.\rho, o_1.\psi)$, $(o_2.\rho, o_{Bob}.\psi)$, and $(o_3.\rho, o_3.\psi)$, for Amy, Bob, and Jack, respectively. $o_1.\psi = \{$ `Audi` $\}$. $o_2.\psi = \{$ `hip-hop` $\}$. $o_3.\psi = \{$ `Audi` $\}$. At time t_1, o_1 and o_2 are updated to $(o_1.\rho', o_1.\psi')$ and $(o_1.\rho', o_2.\psi')$, respectively, where $o_1.\psi' = \{$ `Audi`, `steak` $\}$ and $o_2.\psi' = \{$ `hip-hop`, `sushi` $\}$. The three shops are represented by three queries q_1, q_2, and q_3. $q_1.\psi = \{$ `hip-hop` $\}$. $q_2.\psi = \{$ `sushi` $\}$. $q_3.\psi = \{$ `Audi` $\}$.*

Although we consider static queries here, our method can be easily extended to support dynamic queries, because updating a query is equivalent to deleting a previous query and inserting a new one. For brevity, we call a dynamic spatial-keyword object an *object*, and a spatial-keyword query a *query*.

To evaluate the relevance of an object o to a query q, our scoring function is defined as follows.

Definition 3 (Spatial-Keyword Similarity) *Given an object o and query q, the spatial-keyword similarity of o w.r.t. q is* [2]

$$SimST(o,q) = q.\alpha \cdot SimS(o.\rho, q.\rho) + (1 - q.\alpha) \cdot SimT(o.\psi, q.\psi). \quad (1)$$

For simplicity, we also call $SimST(o,q)$ the *score* between o and q. It is a linear combination of spatial similarity $SimS$ and keyword similarity $SimT$. $SimS$ is calculated by the normalized Euclidean similarity:

$$SimS(o.\rho, q.\rho) = 1 - \frac{Dist(o.\rho, q.\rho)}{maxDist}, \quad (2)$$

where $Dist(\cdot,\cdot)$ measures the Euclidean distance and $maxDist$ is the maximum Euclidean distance in the space. $SimT$ is

[2]The scoring function is also used in [15].

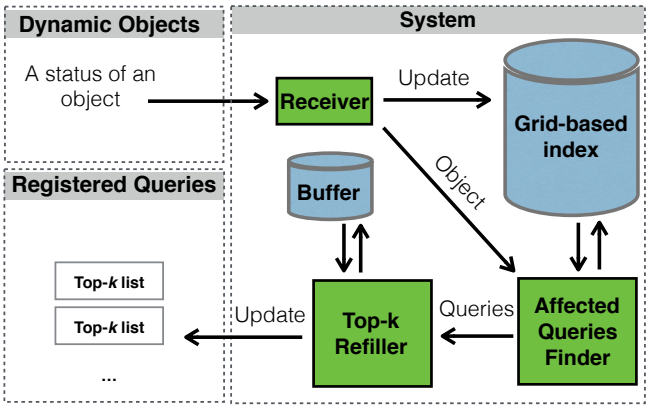


Figure 2: System overview.

calculated using the keywords in $o.\psi$ and $q.\psi$, weighted by tf-idf:

$$SimT(o.\psi, q.\psi) = \sum_{w \in o.\psi \cap q.\psi} wt(o, w) \cdot wt(q, w), \quad (3)$$

where $wt(o, w)$ ($wt(q, w)$) denotes the tf-idf weight of keyword w in o (q). The weights of the keywords in an object or a query are normalized to unit length. As such, $SimT$ exactly captures the cosine similarity between $o.\psi$ and $q.\psi$.

By the above definitions, our problem is defined as follows.

Problem 1 (Continuous Top-k Spatial-Keyword Search on Dynamic Objects) *Given a set of dynamic objects O and a set of queries Q, our goal is to monitor the top-k objects $o \in O$ ranked by descending order of $SimST(o, q)$ for each query $q \in Q$ at each timestamp.*

To ensure that the results are relevant, we require that every object in the top-k results of a query must contain at least one common keyword with the query. In addition, our method can be extended to support the case when each query q has a query-specified k, though a global k is assumed here. Figure 1 shows an example of the top-1 results for three queries at two timestamps.

4. SYSTEM OVERVIEW AND INDEX

We focus on in-memory solutions to continuous top-k spatial-keyword search on dynamic objects. Figure 2 shows the overview of our system. Objects are queries are indexed in a grid or a quadtree. For ease of exposition, we describe our method on top of a grid index in the rest of the paper, unless otherwise noted. Each objects is indexed in the cell in which the object is located, whereas a query q can be indexed in multiple cells, so as to quickly find the affected queries when an object is updated. We dedicate the details of query indexing in the next section.

We assume that in the initial state of the system, there have already been a set of objects and a set of queries, and the top-k results of the queries have already been computed. The initialization of top-k can be done using the grid index; i.e., for each query, we search the cell in which the query resides and then nearby cells. To speed up this process, we can bound the keyword similarities between the query and the objects in a cell using the maximum weights of the query keywords. Then the overall scores can be bounded to prune the cells that are guaranteed not to contain any top-k object of the query. Since we focus on solving the dynamic case of the problem, we omit the details of the initialization in this paper. When the status of an object changes, the grid index is updated. Then the system updates the top-k results for the set of queries. Two modules are designed for efficient query processing: (1) an **affected query finder** that finds the set of affected queries (by "affected", we mean at least one top-k object of this query is replaced or changes its similarity w.r.t. the query), and (2) a **top-k refiller** that updates top-k results of these affected queries.

5. AFFECTED QUERY FINDER

When the system receives a new status of a dynamic object, the affected query finder (AQF) finds the affected queries whose top-k results or their scores need to be updated. We use o and o' to denote the old status and the new status of an object, and they correspond to two contiguous timestamps t and t', respectively. Updating from o to o' affects two sets of queries, denoted by Q_o and $Q_{o'}$. Q_o is the set of queries of which o is a top-k result. $Q_{o'}$ is the set of queries of which o' is a top-k result. It is easy to see that we only need to consider Q_o and $Q_{o'}$, and other queries in Q can be safely excluded. Next we introduce our method to compute Q_o and $Q_{o'}$.

For Q_o, since we have already obtained the top-k results for all the queries at the old timestamp t, an inverted index can be employed to map o to the list of queries of which o is a top-k result. In contrast, computing $Q_{o'}$ is much more challenging. In order to quickly prune queries that are guaranteed not in $Q_{o'}$, we propose the notion of *influential circle* (IC), which bounds the region in which an object may become a top-k result of a query q. In other words, if an object is outside the circle, it cannot be a top-k result of q, regardless of the keyword similarity. To compute the radius of the IC of q, because $SimT(o.\psi, q.\psi)$, the keyword similarity, is bounded by 1 (due to the normalized weights of the keywords), we may replace $SimT(o.\psi, q.\psi)$ with 1 in Equation 1 and $SimS(o.\rho, q.\rho)$ with the spatial similarity of the k-th object. Then the radius of the IC of q

$$q.r = \frac{1 - kScore(q, t)}{q.\alpha} \cdot maxDist, \quad (4)$$

where $maxDist$ is the maximum distance in the space and $kScore(q, t)$ is the score of the k-th object in q's top-k list. We have the following lemma:

Lemma 1 *Given a new status of an object o' and a query q. Let IC_q denote the IC of q. If $q \in Q_{o'}$, then $o'.\rho \in IC_q$.*

The lemma suggests that we only need to consider the queries whose ICs contain the location of o'. To utilize this lemma, we index q in the cells that overlap q's IC. An example is shown in Figure 3. The IC of q_1 overlaps cells c_1, c_2, c_3, and c_4. Hence q_1 is indexed in the four cells. Then given the new status o', by locating the cell of $o'.\rho$ in the grid (denoted by $o'.c$), we check the queries indexed in this cell. First, we test if the IC of a query contains $o'.\rho$. For the queries that satisfy this condition, we compute the score between o' and q. Then we have $Q_{o'} = \{\, q \mid SimST(o', q) > kScore(q, t) \,\}$. E.g., in Figure 3, suppose $o' = o_1$. Because q_1 is indexed in c_1, the cell where o_1 is located, we compute the distance between q_1 and o_1, and find that o_1 is outside the IC of q_1. So q_1 cannot be a query in $Q_{o'}$.

CIQ [3] and SKYPE [15] also comprise a module for finding affected subscriptions, which are analogous to the queries defined in our problem. For each cell that may contain affected subscriptions (by spatial similarity), they iterate through the

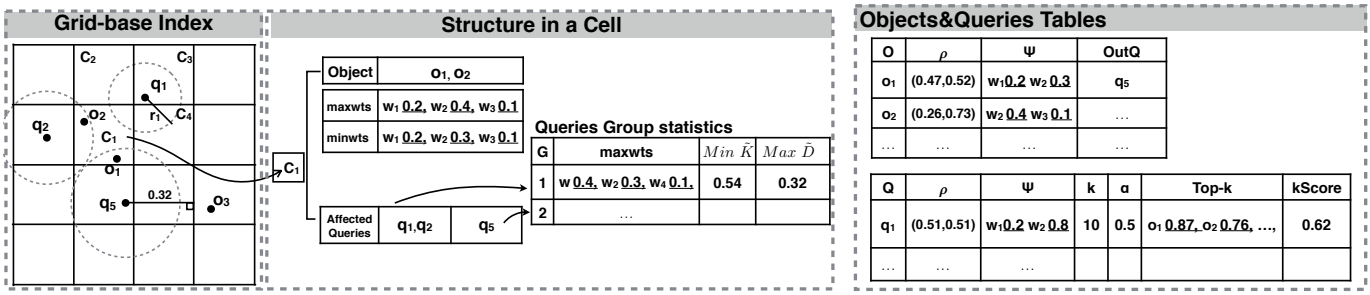


Figure 3: Grid-based index, inner structure of a cell, and tables.

keywords of a geo-textual message (analogous to an object in our problem). For each keyword, the queries having this keyword are identified by an inverted index. SKYPE further employs the prefix filtering technique used in set similarity search [1,19] for pruning, so that only keywords with adequately large weights are considered. However, we find that such scheme is not efficient to our problem: because the set of keywords can be small, it is likely that an object has only frequent keywords, e.g., { `restaurant`, `car` }, which leads to a large number of queries to be scanned. The performance is deteriorated when keyword update exists (note that this does not occur in SKYPE's problem settings), since inserting a frequent keyword is common. In contrast, our affect query finder is based on a spatial pruning strategy without needing an inverted index for keyword-query lookup, hence circumventing the inefficiency caused by frequent keywords.

We also note that some pruning techniques of SKYPE are orthogonal to our IC-based method and can be adapted to our problem to further improve the pruning performance. Next we discuss such adaptation.

First, the spatial similarity of o' to q can be upperbounded using the cell of o'. By Equation 2, we have

$$SimSUB(o'.c, q) = 1 - \frac{minD(o'.c, q.\rho)}{maxDist}, \quad (5)$$

where $minD(o'.c, q)$ is the minimum distance between q and $o'.c$; e.g. in Figure 3, $minD(o_3.c, q_5) = 0.32$. Then we can infer a keyword similarity threshold $T_\theta(o'.c, q)$ with $SimSUB(o'.c, q)$ and Equation 1:

$$T_\theta(o'.c, q) = \frac{kScore(q,t)}{1-q.\alpha} - \frac{q.\alpha}{1-q.\alpha} \cdot SimSUB(o'.c, q). \quad (6)$$

We may compare the keyword similarity with this threshold instead of calculating the whole score. A query is safely discarded if the keyword similarity is not above the threshold.

Second, comparing o' with the indexed queries in a one-by-one fashion is inefficient. The queries indexed in the same cell can be divided into groups, and then we can use a threshold of each group for pruning. To simplify the notations in Equation 6, we divide the threshold into two parts: $\frac{kScore(q,t)}{1-q.\alpha}$ and $\frac{q.\alpha}{1-q.\alpha} \cdot SimSUB(o'.c, q)$, and let $\tilde{K}$ and $\tilde{D}$ to denote the two parts, respectively. Then for a group of queries, denoted by g, we have a keyword similarity between o' and g:

$$T_\theta(o'.c, g) = Min_{q \in g}\{\tilde{K}\} - Max_{q \in g}\{\tilde{D}\}, \quad (7)$$

meaning that for any query in g, if it is in $Q_{o'}$, its keyword similarity to o' must exceed this threshold. To utilize this threshold, we may upperbound the keyword similarity between o' and the queries in g as follows.

$$maxT(g, o') = SimT(g.maxwts, o'.\psi), \quad (8)$$

Algorithm 1 AQF (Affected Query Finder)

Input: o, o'
Output: Q_o, $Q_{o'}$
1: Compute Q_o by inverted index.
2: $Q_{o'} = \emptyset$
3: **for** each group $g \in G$ of $o'.c$ **do**
4: **if** $maxT(g, o') \leq g.T_\theta$ **then**
5: **for** each query $q \in g$ **do**
6: **if** $Dist(o'.\rho, q.\rho) < q.r$ **then**
7: **if** $SimST(o', q) > kScore(q, t)$ **then**
8: $Q_{o'} = Q_{o'} \cup \{ q \}$
9: **return** $Q_o, Q_{o'}$

where $g.maxwts$ denotes the maximum weights of the keywords contained by the queries of g. Since queries are static, $g.maxwts$ is static and can be computed when we create groups for queries. Then the following lemma holds.

Lemma 2 *Given a new status of an object o' and a group of queries g. If $maxT(g, o') < T_\theta(o'.c, g)$, then $g \cap Q_{o'} = \emptyset$.*

For the sake of pruning power, the queries that have similar $T_\theta(o'.c, q)$ values should be put in the same group. In Equation 7, $\tilde{D}$ is static w.r.t. q because the location and the α value of a query never changes. Thus, we choose to group the indexed queries in the same cell by their $\tilde{D}$ values. A quantile-based method is used to partition the domain of $\tilde{D}$ for grouping, in line with the grouping method used in SKYPE. Figure 3 shows an example of the query groups in a cell.

Algorithm 1 gives the pseudo-code of our algorithm for the AQF. Q_o is compute through inverted index access (Line 1). To compute $Q_{o'}$, for each group of queries in $o'.c$, we check the condition in Lemma 2 for pruning (Line 4). For the groups that pass this filter, we compute the distance between each query and o', checking if o' is inside the IC of the query (Line 6). If so, we compute the score between o' and q and add the query to $Q_{o'}$ if the score exceeds the k-th object's score (Lines 7 – 8). The worst-case time complexity of the algorithm is $O(|Q_{o'.c}|)$, where $Q_{o'.c}$ denotes the set of queries indexed in $o'.c$.

6. TOP-K REFILLER

After identifying Q_o and $Q_{o'}$ by the AQF, we update the top-k results for the queries in the two sets. For the queries $Q_{o'}$, this is straightforward, because we only need to insert o' into the query's top-k list and delete the previous k-th object from the list. For the queries in Q_o, since o' may get out of the top-k list and we need to refill the list, the update is much more challenging. A sequential scan of all the objects in O is prohibitive for online services. To address this issue, we first

propose a baseline solution which maintains a sorted cell list (CL) to refill the top-k lists for Q_o queries. Then we propose an improved solution based on a partial cell list (PCL). The PCL method maintains only a few cells and thus avoids many redundant operations in the CL method.

6.1 The Cell List (CL) Method

The top-k reevaluation was studied in kmax [21] and SKYPE [15]. To address the technical challenges of the problems studied in [21] and [15], both methods maintain objects in a buffer for fast reevaluation. However, keeping objects in a buffer is inefficient in our problem because objects are dynamic, incurring frequent buffer maintenance and hence considerable overhead for online processing. For this reason, rather than keeping objects in a buffer, we propose to use a *sorted cell list* that maintains the *cells* in the grid index. It has the following advantages: (1) the number of cells is much smaller than the number of objects; (2) the bound of spatial similarity from a cell to a query is static; and (3) despite dynamic objects in a cell, the keyword similarities from the objects in a cell to a query can be bounded without frequent changes. Next we introduce our CL method for top-k reevaluation.

In order to bound the keyword similarities, we store additional information in the grid index. For each cell, we collect the distinct keywords that appears in the objects of this cell, and store a value for each keyword w, denoted by $maxwts$, representing the maximum weight of w among all this cell's objects containing w. It is used to bound the keyword similarities from the cell's objects to a query. Let $c.maxwts$ denote the set of the $maxwts$ values of the keywords that appear in c.

Example 2 *In Figure 3, there are two objects, o_1 and o_2, in c_1. Suppose there are three distinct keywords in o_1 and o_2, denoted by w_1, w_2, and w_3. Then $c_1.maxwts = \{0.2, 0.4, 0.1\}$.*

Then we are able to upperbound the scores of the objects in a cell.

$$maxscore(c, q) = q.\alpha \cdot SimSUB(c, q.\rho) + (1 - q.\alpha) \cdot SimT(c.maxwts, q.\psi). \quad (9)$$

The $SimSUB$ value is computed by Equation 5. The $SimT$ value is computed by iterating through the keywords in $q.\psi$. For each q, we store in a list (denoted by $q.CL$) all the cells in the grid, and sort them by decreasing order of $maxscore(c, q)$. An example is shown in Figure 4. Since the $maxscore(c, q)$ value may change due to dynamic objects, we implement $q.CL$ as a binary-tree-like data structure [3].

Then we design an algorithm by leveraging the sorted cell lists. The pseudo-code is shown in Algorithm 2. For the queries in Q_o and $Q_{o'}$, we first compute the new $maxscore(c, q)$ values for $o.c$ and $o.c'$ (Line 2). Then the top-k results are updated for each query. For the queries in $Q_{o'}$, the update is easy: we only insert o' into the queries' top-k lists (Line 4). For the queries in Q_o, we scan each cell in $q.CL$ and compare its $maxscore(c, q)$ with the current k-th object's score (Line 7). If the former is no greater than the latter, the algorithm terminates because the $maxscore(c, q)$ values of the unseen cells are even smaller. Otherwise, we scan each object in this cell and update the top-k list of the query (Lines 9 – 11). Note that for the queries that appear in both Q_o and $Q_{o'}$, we scan skip the procedure of processing $q.CL$, because their top-k results have been updated in Line 4. Hence we use a set difference in Line 5.

[3]In our experiments, we use std::map in C++, which is implemented as a red-black tree.

Algorithm 2 CL

Input: o, o', Q_o, $Q_{o'}$
Output: top-$k(q, t')$ for $q \in Q_o \cup Q_{o'}$
1: **for** each $q \in Q_o \cup Q_{o'}$ **do**
2: Update $maxscore(o.c, q)$ and $maxscore(o'.c, q)$ in $q.CL$
3: **for** each $q \in \{Q_{o'}\}$ **do**
4: top-$k(q, t')$.insert(o')
5: **for** each $q \in \{Q_o \backslash Q_{o'}\}$ **do**
6: **for** each $c \in q.CL$ **do**
7: **if** $maxscore(c, q) \leq kScore(q, t)$ **then**
8: break
9: **for** each $o'' \in c$ **do**
10: **if** $SimST(o'', q) > kScore(q, t)$ **then**
11: top-$k(q, t')$.insert(o'')

6.2 The Partial Cell List (PCL) Method

The main drawback of the CL method is that it maintains all the cells in the grid, while it is likely that most cells can be ignored when we update the top-k list. It can be seen that when o changes to o', for any $q \in Q_o$, at most one object in the top-k list of q changes. Moreover, we have the following observation.

Observation 1 *Consider an old status o and a new status o' of an object, which correspond to two contiguous timestamps t and t', respectively. Let $q.o_{k+1}$ denote the $(k+1)$-th result of q at time t. For any $q \in Q_o$, top-$k(q, t') \setminus$ top-$k(q, t) = \{o'\}$ or $\{q.o_{k+1}\}$.*

In other words, we only need to compare o' with the $(k+1)$-th object of q at time t, and pick the one with the higher score as the new top-k result of q. Based on this observation, we devise an improved method for top-k reevaluation by maintaining a partial cell list (PCL) for each q such that the $(k+1)$-th object of q at time t is guaranteed to reside in one of these cells. Note that we do not simply maintain the $(k+1)$-th result since this incurs very frequent update and compromises the efficiency; nor we use the method in [12, 15] to keep a list of objects whose scores are above a threshold, because the objects are dynamic in our problem and a reordering of the list is required whenever an object changes its status. Instead, we maintain cells, in which the objects' scores can be bounded and do not change frequently.

For the sake of efficiency, we want to exclude in the PCL of q (denoted by $q.PCL$) the cells that are guaranteed not to contain the $(k+1)$-th result. To achieve this, we compute a lowerbound of the objects' scores in a cell, in addition to the upperbound in the CL method. First, we store in the index the minimum weight of a keyword, denoted by $minwts$. Let $c.minwts$ denote the set of the minimum weights of the keywords that appear in c.

Example 3 *We continue Example 2 for the two objects and three keywords in c_1. $c_1.minwts = \{0.2, 0.3, 0.1\}$.*

Then we compute the lowerbound of the scores of the objects in a cell c:

$$minscore(c, q) = \alpha \cdot SimSLB(c, q.\rho) + (1 - \alpha) \cdot SimTLB(c.minwts, q.\rho). \quad (10)$$

$SimSLB$ is given by

$$SimSLB(c, q) = 1 - \frac{maxD(c, q.\rho)}{MaxDist}, \quad (11)$$

where $maxD(c, q.\rho)$ represents the maximum Euclidean distance between an object in c to q. $SimTLB$ is given by

$$SimTLB(c.minwts, q.\rho) = \min_{w \in q.\psi \cap c.minwts} wt(q.w) \cdot wt(c.w). \quad (12)$$

Then we determine the cells that can be excluded from $q.PCL$. We have the following lemmata.

Lemma 3 *Given a cell c, if $minscore(c, q) > kScore(q, t)$, then $q.o_{k+1} \notin c$.*

PROOF. Because $minscore(c, q) > kScore(q, t)$, for any $o_i \in c$, the score of o_i at time t is greater than $kScore(q, t)$, and thus greater than the score of $q.o_{k+1}$. Therefore, $q.o_{k+1} \notin c$. □

Lemma 4 *Let $maxminS = \max\{ minscore(c_i, q) \}$, $c_i \in grid$ and $maxscore(c_i, q) < kScore(q, t)$. Given a cell c, if $maxscore(c, q) < maxminS$, then $q.o_{k+1} \notin c$.*

PROOF. We prove by contradiction and assume $q.o_{k+1} \in c$. Let $c' = \arg\max_{c_i} minscore(c_i, q), c_i \in grid$ and $maxscore(c_i, q) < kScore(q, t)$, i.e., the cell that yields $maxminS$. Because $maxscore(c, q) < maxminS = minscore(c', q) \leq maxscore(c', q)$, $\exists o_i \in c'$, s.t. the score of o_i is greater than that of $q.o_{k+1}$. Because $maxscore(c', q) < kScore(q, t)$, the score of o_i is smaller than that of the k-th object, meaning that either o_i is $q.o_{k+1}$ or o_i's score is even smaller than that of $q.o_{k+1}$. This contradicts that o_i's score is greater than that of $q.o_{k+1}$. □

Intuitively, the two lemmata state that if a cell whose objects' scores are too high or too low, then we do not keep it in $q.PCL$. Thus, we only keep in $q.PCL$ the cells such that $minscore(c, q) \leq kScore(q, t)$ and $maxscore(c, q) \geq maxminS$. **FIXME!!!** The lowerbound is initialized and will not change until $q.PCL$ is recreated. This guarantees that $q.PCL$ has the cell with the $(k+1)$-th result:

Lemma 5 *$\exists c \in q.PCL$, s.t., $q.o_{k+1} \in c$.*

PROOF. We prove by contradiction. If $\nexists c \in q.PCL$, s.t., $q.o_{k+1} \in c$, then the cell that contains $q.o_{k+1}$, denoted by c', satisfies either $minscore(c, q) > kScore(q, t)$ or $maxscore(c, q) < maxminS$. This either contradicts Lemma 3 or Lemma 4. □

Example 4 *Figure 4 gives an illustration of CL and PCL buffers. In this Figure, the horizontal axis shows the score distribution of all cells (c_1, c_2, ..., c_n) w.r.t a specific q. A segment represents the score range of a cell. The left side is the maxscore while the right side is the minscore. CL is a cell list sorted by maxscore. Therefore. the order of $q.CL$ in this example is: $\{c_5, c_2, c_1, c_6, ..., c_n\}$. For PCL, $q.PCL.up$ is initialized by the current $kScore(q)$. On the right side of $kScore(q)$, cell c_1 has the maximum minscore. Thus, $q.PCL.low$ will be initialized as $minscore(c_1, q)$. Then $q.PCL$ contains the $\{c_2, c_1, c_6\}$ overlapping the range $(q.PCL.up, q.PCL.low)$.*

For each query, the corresponding PCL is initialized based on Definition **??**. When a top-k list needs to be reevaluated, the candidate object can be searched from PCL and refilled to this top-k list. The top-k reevaluation is much more efficient than using the CL because the only top-1 search is conducted from fewer cells. To guarantee that all PCLs always contain the candidate object w.r.t the corresponding queries, we propose a sophisticated strategy to maintain PCL.

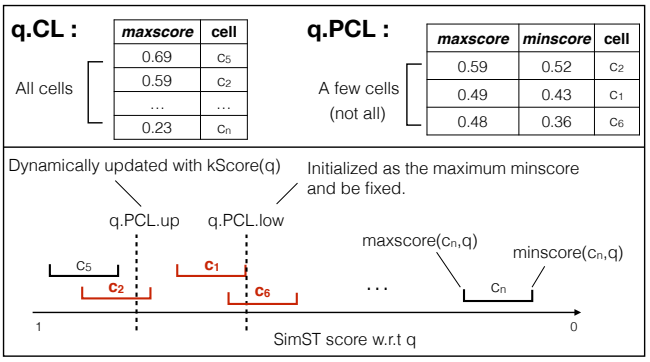


Figure 4: Examples of CL (Section 6.1) and PCL (Section 6.2) buffers.

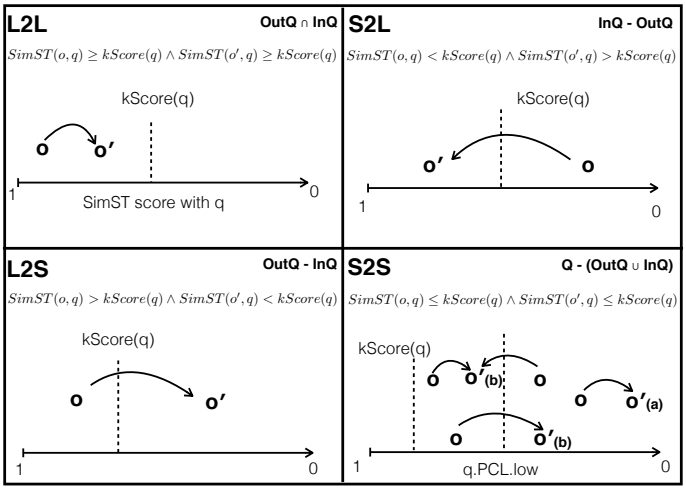


Figure 5: Four cases of a dynamic object and a query.

6.2.1 PCL Maintenance

We divide the situations between a dynamic object and a query into the following four cases. Figure 5 shows images and summarizes the four cases: **L2L** (large to large), **S2L** (small to large), **L2S** (large to small) and **S2S** (small to small). Because a dynamic object may affect the cells in PCL, we should maintain PCL carefully to ensure it always contains the $(k+1)$-th object to refill. We propose a sophisticated maintenance process. Table 3 summarizes the operations of these cases.

L2L and S2L. The cases of L2L and S2L are discussed together since they have a common PCL maintenance. In the cases of L2L, both o and o' are in the top-k. Thus, the order of the objects outside top-k is not changed, and the $(k+1)$-th object remains in PCL. Therefore, PCL still contains the candidate object. We just *check* the changing cells $o.c$ and $o'.c$ with PCL. Algorithm 3 describes the procedure of *check*. In Algorithm 3, each input cell, according to their new *maxscore* and *minscore*, will be deleted (Lines 2-3) and re-inserted (Lines 4-5) into PCL. For the case of S2L, o' is added into top-k from the outside. The previous kth object (denoted as o_k) will become the $(k+1)$ one, so we should add the $o_k.c$ into PCL to ensure the candidate object. Since $o.c$ and $o'.c$ also require a *check*, we need *check* $\{o.c, o'.c, o_k.c\}$. In conclusion, we employ a *check* operation to maintain PCL in the cases of L2L and S2L.

Lemma 6 *Given a dynamic object and a query. If they are*

Algorithm 3 q.PCL.check

Input: $Cells$
1: **for** each cell $c \in Cells$ **do**
2: **if** $c \in PCL$ **then**
3: delete c from q.PCL
4: **if** $(maxscore(c,q) > q.PCL.low)$ and $(minscore(c,q) < kScore(q))$ **then**
5: $q.PCL.insert(c)$

Algorithm 4 $q.PCL.trim$

Input: $kScore(q)$, $q.PCL.Low$
1: $PCL_{new} = \emptyset$
2: **for** each $c \in q.PCL$ **do**
3: **if** $(maxscore(c,q) > q.PCL.low)$ and $(minscore(c,q) < kScore(q))$ **then**
4: $PCL_{new}.insert(c)$
5: $q.PCL = PCL_{new}$

in the cases L2L and S2L, we need to maintain PCL with the operation of check.

L2S. In the situation of L2S, we must *search* the top-1 candidate object o_{cand} from PCL. Then the scores $SimST(o', q)$ with $SimST(o_{cand}, q)$ are compared to determine whether to refill o_{cand} or not. Then, we should *trim* the PCL with a new score range $(kScore(q)', q.PCL.low)$ where $kScore(q)'$ denotes the updated k-th score. Algorithm 4 gives the details of the operation *trim*, which filters the cells that are not within the input score range (Lines 2-4). Sometimes PCL may become empty (no object) after *trim*. Then, PCL must be recreated.

Lemma 7 *Given a dynamic object and a query of case L2S, we need to maintain PCL with the operations of check and trim. Also, if PCL has empty candidates after trimming in case L2S, PCL must be recreated.*

S2S. S2S is divided into two sub-cases: S2S.a and S2S.b. In S2S.a, both o and o' are on the outside of $q.PCL.low$, so PCL does not need to be maintained. In S2S.b, we need to *check* $o.c$ and $o'.c$ with PCL. Similar to L2S, we will recreate PCL if it becomes empty.

Lemma 8 *Given a dynamic object and a query, if they are in case S2S.a, PCL does not need to be maintained. On the other hand in S2S.b, PCL must be maintained with the* check *operation. If PCL has no candidates, PCL must be recreated.*

Algorithm 5 gives the proposed GPCL method. It maintains the top-k and PCL buffer for the 4 cases illustrated in Figure 5. In L2L and S2L (Lines 1-8), the top-k lists are only updated with the upcoming o' (Lines 3, 7). PCL will be checked with Algorithm 3 (Lines 4,8) based on Lemma 6. In L2S (Lines 9-20), a candidate object will be searched and re-fill to top-k (Lines 11-15). PCL will be trimmed with Algorithm 4 (Line 17) based on Lemma 7. In S2S (Lines 21-27), PCL will be checked based on Lemma 8. Note that when PCL becomes empty, it will be re-created (Lines 19, 27).

7. GRID SIZE TUNING

Compared to CL, PCL is advantageous with regards to a (k+1)-th maintenance. On the other hand, if PCL is empty, it must be recreated from all cells, which is an expensive operation

Algorithm 5 GPCL

Input: o, o', Q_o, $Q_{o'}$
1: // **L2L**
2: **for** each $q \in Q_o \cap Q_{o'}$ **do**
3: Update $top\text{-}k(q)$ with o'.
4: $q.PCL.check(\{o.cell \cup o'.cell\})$
5: // **S2L**
6: **for** each $q \in Q_{o'} - Q_o$ **do**
7: Update $top\text{-}k(q)$ with o'.
8: $q.PCL.check(\{o.cell \cup o'.cell \cup k.cell\})$
9: // **L2S**
10: **for** each $q \in Q_o - Q_{o'}$ **do**
11: o_{cand} = Retrieve Top-1 from $q.PCL$
12: **if** $SimST(o', q) > SimST(o_{cand}, q)$ **then**
13: Update $top\text{-}k(q)$ with o'.
14: **else**
15: Update $top\text{-}k(q)$ with o_{cand}.
16: $q.PCL.check(\{o.cell \cup o'.cell\})$
17: $q.PCL.trim(kScore(q)', q.PCL.low)$
18: **if** $q.PCL$ is empty **then**
19: $q.PCL.recreate$
20: // **S2S**
21: **for** each $q_i \in Q - Q_o \cup Q_{o'}$ **do**
22: **if** $SimST(o, q) < q.PCL.low$ and $SimST(o', q) < q.PCL.low$ **then**
23: **continue**
24: **else**
25: $q.PCL.check(\{o.cell \cup o'.cell\})$
26: **if** $q.PCL$ is empty **then**
27: $q.PCL.recreate$

Table 3: Summary of the operations for PCL maintenance.

	Operation on PCL			
Case	Search	Check	Trim	Re-create
L2L	-	✓	-	-
S2L	-	✓	-	-
L2S	✓	✓	✓	✓(if empty)
S2S.a	-	-	-	-
S2S.b	-	✓	-	✓(if empty)

$(O(n^2 log n^2))$. Therefore, there is a trade-off between the number of cells and recreation. Fewer cells in PCL realize an efficient search and maintenance, but lead to a higher probability that an expensive recreation will be triggered. Obviously, the number of cells in PCL depends on the number of cells n^2 in the grid. We conduct a theoretical analysis to build a cost model that balances this trade-off by estimating the best value of n^2. The total expecting cost is the sum of the expected costs of each operation.

$$E[total] = E[check] + E[trim] + E[search] + E[recreate] \tag{13}$$

The expected cost of an operation can be calculated by multiplying the probability to the number of the similarity computation. According to Table 3, the expected costs of all operations can be calculated by the equations below. Here, $P(X)$ means the probability that event X happens, PCL_c represents the cells number in PCL, and PCL_o represents the object's number in PCL.

$$E[check] = (1 - P(S2S.a)) \cdot log(PCL_c) \tag{14}$$

$$E[trim] = P(L2S) \cdot PCL_c \tag{15}$$

$$E[search] = P(L2S) \cdot log(PCL_o) \quad (16)$$

$$E[recreate] = P(PCL.empty) \cdot n^2 \cdot logn^2 \quad (17)$$

To estimate $E[total]$, we need to find the probabilities of $P(S2S.a)$, $P(L2S)$ and $P(PCL.empty)$. We implement a theoretical analysis based on the following assumption. The queries and objects follow a uniform distribution in [0,1) space. With n^2 cells, the length of a cell is $l = \frac{1}{n}$, and the average number of objects in a cell is $\frac{|O|}{n^2}$, where $|O|$ is the size of the object set. For an object, we assume that a spatial (maximum) step in space is δ_s. To evaluate the similarities on spatial and keywords, we convert the value on one to the other. δ_t is calculated by computing the change on the keyword similarity and converting it to space. We use w_q and w_o to represent the average number of keywords in an object and a query, respectively. The change in the keyword similarity is:

$$SimT(o'.\psi, q.\psi) - SimT(o.\psi, q.\psi) = \sum_{w \in o.\psi \cap q.\psi} wt(q.w) \cdot (wt(o'.w) - wt(o.w)) \approx \frac{1}{w_q^2 \cdot w_o^2} \quad (18)$$

By recalling that the distance in [0,1) space ranges is from 0 to $\sqrt{2}$, while the similarity on the keywords is the cosine value, we can convert the change of keyword similarity to that on the space by multiplying the ratio between the two ranges as

$$\delta_t = \frac{\sqrt{2}}{w_q^2 \cdot w_o^2}. \quad (19)$$

Then we estimate the probabilities. Recall that L2S is the case where an object belongs to top-k and changes out of top-k. As shown in Figure 6a, $P(L2S)$ can be estimated as the ratio of the areas, which is expressed as:

$$P(L2S) = \frac{S_{\delta_1}}{S_k + S_{\delta_1}} \cdot S_k = \pi {r_k}^2 - \frac{\pi r_k^4}{(r_k + \delta_s + \delta_t)^2} \quad (20)$$

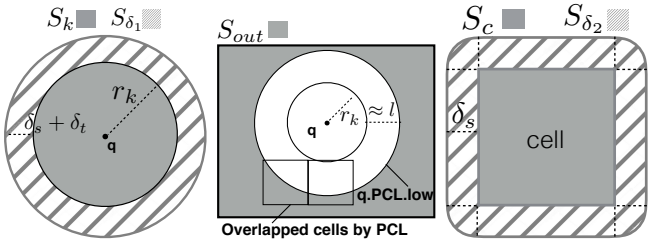
Since we assume that all points follow a uniform distribution, the ratio of the area between the top-k circle and the whole space represents the ratio of the object's numbers. Hence, the radius r_k in Equation (20) can be calculated as:

$$\frac{\pi {r_k}^2}{1} = \frac{k}{|O|} \implies r_k = \sqrt{\frac{k}{\pi |O|}} \quad (21)$$

According to Figure 6b, P(S2S.a) is computed based on the areas in a similar way.

$$P(S2S.a) = {S_{out}}^2 = (1 - \pi(r_k + l)^2)^2 \quad (22)$$

The number of cells in PCL, PCL_c, is estimated by the number of cells that overlap the circle of q.PCL.low. As shown in Figure 6b, the number of overlapped cells is approximately the value of the perimeter divided by the side length of a cell (l). That is, $PCL_c = \frac{2\pi(r_k+l)}{l}$. Then we can calculate the objects number $PCL_o = PCL_c \cdot \frac{|O|}{n^2}$. Because PCL has PCL_o objects, the probability that PCL only has one object can be simply approximated as $\frac{1}{PCL_o}$. When PCL contains only one object, PCL may become empty when this object moves out of its cell. Note that moving out of a cell depends only on δ_s. Figure 6c shows the areas of this situation. In conclusion, $P(PCL.empty)$ is calculated as:



(a) L2S. (b) S2S.a. (c) Out of a cell.

Figure 6: Area of each case.

$$P(PCL.empty) = \frac{1}{PCL_o} \cdot \frac{S_{\delta_2}}{S_{\delta_2} + S_c} = \frac{1}{PCL_o} \cdot \frac{4l\delta_s + \pi\delta_s^2}{4l\delta_s + \pi\delta_S^2 + l^2} \quad (23)$$

Eventually, with the parameters $|O|$, k, δ_s, w_q and w_o, the only variable in Equation (13) is n. We used the gradient descent to figure out the n by computing the extreme value to minimize $E[total]$.

8. BATCH PROCESSING

If an application requires a massive number of object to be updated frequently, it may become inefficient to process the new status of the objects one by one. In such cases, batch processing is a good choice to combine the reduplicative updating top-k list with common queries. Our methods can be extended to support a batch process easily. Suppose that we received a batch of objects and aim to carry out a batch process. In *affected queries finder* module, we retrieve $Q_{o'}$ and Q_o for a specific object when processed singly. To identify $Q_{o'}$ and Q_o to build a batch of objects, we need to count the number of times of a query belongs to $Q_{o'}$ and Q_o in this batch. If the times of a query belongs to $Q_{o'}$ is larger or equal to the times that of Q_o, then we mark this query as a $Q_{o'}$ query. Otherwise, it should be a Q_o query. In *Top-k refiller* module, we need to initialize PCL to maintain k-candidate objects so that we can utilize a top-k search on the PCL buffer. The PCL buffer can be maintained with a set of cells in which a batch of objects arrive in (depart from). The operations and methodologies for PCL maintenance in Section 6.2 also support a batch of cells directly.

On the other hand, aiming at a real-time processing problem, we need to consider the response time. It is well-known that large size of batch would result in a faster average processing time, it is unrealistic to let the client wait too long when filling up the batch. Therefore, a balance needs to be made in practice by considering several items such as computing resource, update rate, and the user requirement. Intuitively, $b_n \sim f(b_r, b_p)$, where b_n, b_r, and b_p are batch size, the user-defined response time and processing time, respectively. And, f maps b_r, b_p to tune the best batch size in practice. As an example, adding b_r to Fig. 16.b we can figure out the appropriate b_n for different processing time.

9. EXPERIMENTS

Table 4: Datasets statistics.

Datasets	YELP	TWITTER	SYN
Data size	1.1M	4.2M	12M
Default # of selected objects	220K	500K	1M
Default # of selected queries	156K	250K	1M
# of keywords	819K	3.5M	819K
# of average keywords	5.9	4.5	3

All algorithms were implemented in C++. All indices, buffers, and algorithms were run on in-memory of a Mac with a 2.2GHz Intel Core i7 CPU and 32GB memory.

9.1 Setting

Dataset. We used two real datasets and one synthetic dataset. Table 4 shows the statistics of the datasets.

YELP is an open source dataset provided by YELP.com[4]. It contains 1.1M of reviews on 156K businesses by 220K users. We set the businesses with their locations and descriptions as queries. For the keyword attributes of queries, we randomly selected 1 to 5 keywords from the description. We set the first review of each user as the initial states of an object. Since the reviews did not contain users' locations, we intuitively set the initial location of the objects as the business location in the review. For the future status of the dynamic objects, we set a simple random walk that randomly ± 0.05 to the previous coordinates. Then we used the contents of other reviews from the same user as the changes in the keyword attribute.

TWITTER is the dataset with 4.2M geo-tag tweets from the United States[5]. In TWITTER, there are 1.2M unique users each of which has at least three geo-tag tweets. First, we selected 100K to 500K random users and used their first tweets as the queries in the experiment. For each query, we set the spatial attribute as their geo-tag. Then select 1 to 5 words from the tweets as the keyword attributes. On the other hand, we randomly selected 200K to 1M from the remaining unique users and initialized the objects with their first geo-tag tweets. The rest of the tweets from selected users were treated as the continuous states of moving objects.

SYN is a synthetic data containing 12M spatial keyword tuples. we used the data of moving points[6] for spatial attributes, which were generated by the BerlinMOD benchmark [7]. For the keyword attributes, we randomly selected 1 to 5 keywords from the TWITTER dataset and assigned them to each point. Objects and queries were selected from the SYN tuples.

Algorithms. For the *affected queries finder* module, we compared the following methods:

- **CIQ.** The block-based inverted file structure in [3].
- **IGPT.** The group pruning techniques in [15].
- **AQF.** The proposed method in this paper with a influence circle and a group pruning technique.

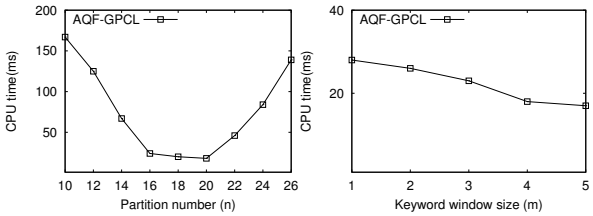
For the *top-k refiller* module, we compared:

- **kmax**. The method with $kmax$ buffer in [21]. For the size of the buffer, we tuned the value of $kmax$ from k to $5k$. The experimental results lead us to set $3k$ as the default value of $kmax$. The tuning result is similar to related work [15].
- **GCL.** The proposed method with CL (Algorithm 2).
- **GPCL.** The proposed method with PCL (Algorithm 5).

[4]https://www.yelp.com/dataset

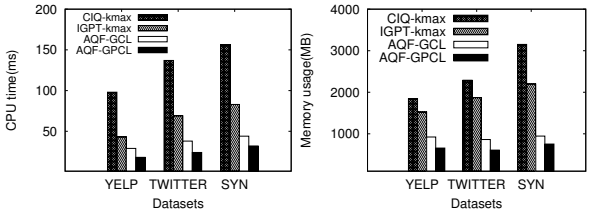
[5]https://datorium.gesis.org/xmlui/handle/10.7802/1166

[6]http://dna.fernuni-hagen.de/secondo/BerlinMOD/BerlinMOD.html



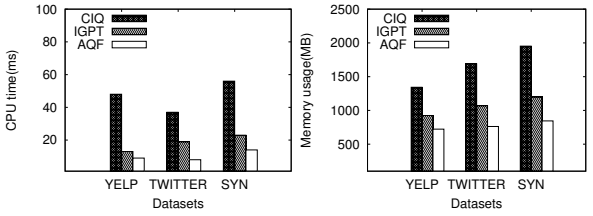
(a) Varying n. (b) Varying m.

Figure 7: TWITTER data.



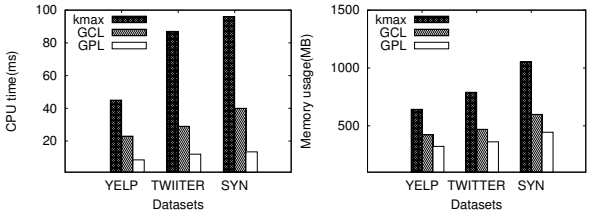
(a) Average processing time. (b) Memory usage.

Figure 8: Overall processing.



(a) Average processing time. (b) Memory usage.

Figure 9: *Affected queries finder* module.



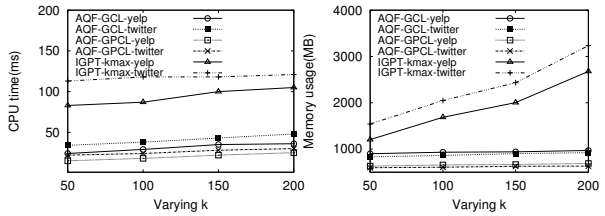
(a) Average processing time. (b) Memory usage.

Figure 10: *Top-k refiller* module.

We randomly selected over 10,000 status of dynamic objects and reported the average processing time and memory usage among different methods. The default result size k was 100 and the parameter α was a random value in (0,1). The default grid size is calculated by Equation (13), e.g., the grid size for the TWITTER data with $k = 100$ is 16^2. The default window size m for the keywords of an object ($o.\psi$) was 1.

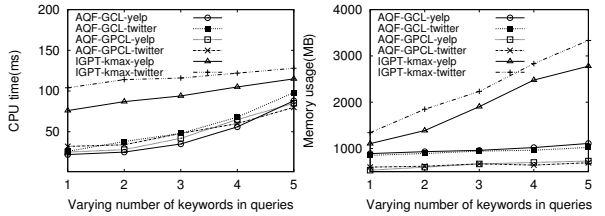
9.2 Experimental Results

Effect on varying n. In Figure 7a, we varied n^2 to observe the processing performance for testing our cost model in Section 7. The optimal theoretical results of n by minimizing Equation (13) is 15.44. According to experimental results, the best value is 20, while the results of 16-18 are close to the optimal value. We conclude that our cost model estimates the n well and helps



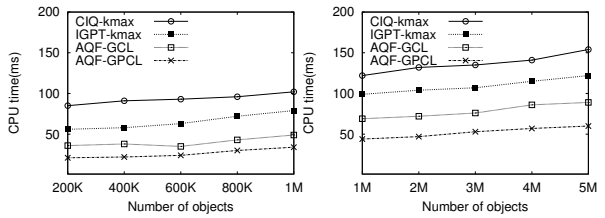
(a) Average processing time. (b) Memory usage.

Figure 11: Varying k.



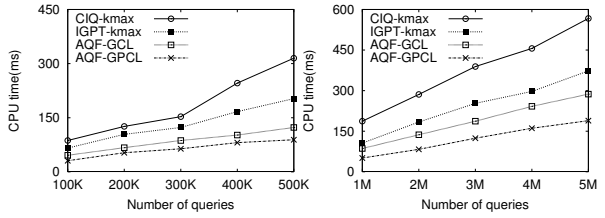
(a) Average processing time. (b) Memory usage.

Figure 12: Varying number of keywords in queries.



(a) TWITTER data. (b) SYN data.

Figure 13: Varying number of objects.



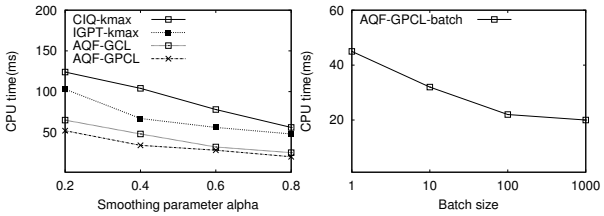
(a) TWITTER data. (b) SYN data.

Figure 14: Varying number of queries.

to optimize the performance.

Effect on varying m. In Figure 7b, each object keeps five recent status of keywords. A Larger m leads a slighter change on the keyword attribute and a better performance on processing. Compared to the situation of keyword changes discretely ($m = 1$), we have less processing time when keywords attribute changes consecutively (i.e., to keep the previous keywords). The reason is that our PCL has a higher probability to offer the candidate results and avoid recreating.

Overall processing. Figure 8 shows the comparison results for the overall processing with the default size of objects and queries shown in Table 4. Specifically, we compared the two proposed methods, AQF-GCL and AQF-GPCL to two related works. Note that AQF-GCL means that we imported the AQF algorithm as the *affected queries finder* module, and utilize the



(a) Varying α. (b) Average processing time.

Figure 15: TWITTER data.

GCL algorithm as the *top-k refiller* module. For the related works IGPT and CIQ, we adjusted their techniques with the $kmax$ method so that they could deal with our problem of dynamic objects. Both of our proposed methods, AQF-GCL and AQF-GPCL, have better performances than the others w.r.t the processing time and memory cost.

Affected queries finder. Figure 9 shows the comparison results of finding affected queries. Our method AQF is at least 1.5 times faster than the other methods. The reason is because IGPT and CIQ required overhead costs on the traversal of the quad-tree index from the root node. In contrast, our grid-based index can index only a few candidate queries and retrieve them directly (o(1) complexity). Moreover, the group pruning technique in AQF also boosts the performance by filtering unnecessary queries. The memory usage of our grid-index for queries is the smallest one (Figure 9b). We only index each query once, while IGPT indexes queries several times with keywords in the inverted files. CIQ indexes queries in multiple nodes of quad-tree.

Top-k refiller. To test our *top-k refiller*, we compare the proposed GCL and GPCL methods with the $kmax$ method. The results include two parts: the processing time of the top-k reevaluation and the maintenance time. According to Figure 10, our proposed methods are at least twice as fast as the $kmax$. The GPCL method is better than GCL since it uses the candidate refilling strategy rather than reevaluating the whole top-k. PCL also balances the trade-off as it only maintains a few cells. Therefore, GPCL has the least memory usage.

Effect on varying k. According to the Figure 11a, Only AQF-GPCL is not influenced by k since the mechanism of GPCL refills the candidate (k+1)-th object rather than reevaluating the whole top-k. From the memory usage of indices in Figure 11b, a large k leads to a bigger index for $kmax$-based methods. However, our proposed methods are unaffected by k since we index the identity of the cells. As we introduced previously, PCL can be seen as a small subset of CL. Hence, AQF-GPCL has a smaller memory cost than AQF-GCL.

Effect on the number of query keywords. Figure 12 shows the processing performance among different methods with a varying number of keywords in queries. Our methods are better than the other methods in all situations for a given number of keywords. The processing time of our proposed methods is close to other methods as the number of keywords increases because many keywords will loosen the bound of spatial keyword similarity of a cell in Equations (9) and (10). Figure 12b shows the memory usage of varying number of keywords in queries. Our methods keep their superiority and AQF-GPCL costs the least memory among all indices (buffers). The inverted-file leads IGPT-based and CIQ-based methods cost more space to index data with more keywords.

Effect on number of objects and queries. Figure 13 shows the effects of varying the cardinality of objects with TWITTER data and SYN data. Since all algorithms keep objects with spatial indexes and implement a buffer to maintain the candidate objects, they are not affected too much by the large size of the objects. However, as the Figure 14 shows, a large size of queries affects the efficacy of all algorithms because more queries are affected by a dynamic object, which subsequently triggers more processes to update the results.

Effect on varying α. Figure 15a shows the results of varying α. All algorithms have better performance in larger α since the pruning power is mainly on the spatial attribute.

Batch process. Figure 15b shows the average processing time of the batch process with different batch sizes on TWITTER data. Note that batch size 1 is the case of the previous singular process of proposed AQF-GPCL. We set that a new status of an object arrives every 10ms (100 geo-tagged tweets/s [3]). The reported results also contain the waiting time that objects arrive in. Obviously, the batch process can enhance the throughput. As discussed in Section 8, we can tune a batch size with this result. e.g.: we input the 500ms as the user maximum tolerate time b_r, and we can get $b_n = 10$ is a proper balanced value for batch size.

10. CONCLUSION

In this paper, we investigate a novel problem of top-k spatial keyword query processing on dynamic objects. We propose a solution system to efficiently search and maintain the top-k results for multiple queries. We employ a grid-based index to handle both dynamic objects and queries. To efficiently detect the affected queries by an object efficiently, we propose a group pruning strategy in our *affected queries finder* module. To maintain the top-k results with a quick-response and low-cost, we propose a sophisticated buffer called the partial cell list (PCL) to efficiently refill the top-k results in our *top-k refiller* module. We also extend the proposed methods to treat the batch process. The experiments confirm that our proposal has a good performance compared with the baselines and related works. As for future work, we plan to research deeply of the batch process on our problem. We also plan to propose a distributed process based solution.

Acknowledgement

We thank Chuan Xiao from Osaka University for commenting and rewriting this paper. This research was partly supported by the program "Research and Development on Real World Big Data Integration and Analysis" of RIKEN, Japan.

11. REFERENCES

[1] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 131–140, 2007.

[2] L. Chen, G. Cong, and X. Cao. An efficient query indexing mechanism for filtering geo-textual data. In *SIGMOD*, pages 749–760, 2013.

[3] L. Chen, G. Cong, X. Cao, and K. Tan. Temporal spatial-keyword top-k publish/subscribe. In *ICDE*, pages 255–266, 2015.

[4] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[5] G. Cong and C. S. Jensen. Querying geo-textual data: Spatial keyword queries and beyond. In *SIGMOD*, pages 2207–2212, 2016.

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

[7] C. Düntgen, T. Behr, and R. H. Güting. Berlinmod: a benchmark for moving object databases. *VLDB J.*, 18(6):1335–1368, 2009.

[8] L. Guo, J. Shao, H. H. Aung, and K. Tan. Efficient continuous top-k spatial keyword queries on road networks. *GeoInformatica*, 19(1):29–60, 2015.

[9] W. Huang, G. Li, K. Tan, and J. Feng. Efficient safe-region construction for moving top-k spatial keyword queries. In *CIKM*, pages 932–941, 2012.

[10] G. Li, Y. Wang, T. Wang, and J. Feng. Location-aware publish/subscribe. In *KDD*, pages 802–810, 2013.

[11] A. R. Mahmood and W. G. Aref. Query processing techniques for big spatial-keyword data. In *SIGMOD*, pages 1777–1782, 2017.

[12] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

[13] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.

[14] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In *SSTD*, pages 205–222, 2011.

[15] X. Wang, Y. Zhang, W. Zhang, X. Lin, and Z. Huang. SKYPE: top-k spatial-keyword publish/subscribe over sliding window. *PVLDB*, 9(7):588–599, 2016.

[16] X. Wang, Y. Zhang, W. Zhang, X. Lin, and W. Wang. Ap-tree: efficiently support location-aware publish/subscribe. *VLDB J.*, 24(6):823–848, 2015.

[17] D. Wu, M. L. Yiu, and C. S. Jensen. Moving spatial keyword queries: Formulation, methods, and analysis. *ACM Trans. Database Syst.*, 38(1):7:1–7:47, 2013.

[18] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *ICDE*, pages 541–552, 2011.

[19] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15:1–15:41.

[20] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.

[21] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[22] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.

[23] B. Zheng, K. Zheng, X. Xiao, H. Su, H. Yin, X. Zhou, and G. Li. Keyword-aware continuous knn query on road networks. In *ICDE*, pages 871–882, 2016.