Continuous Top-k Spatial-Keyword Search on Dynamic Objects

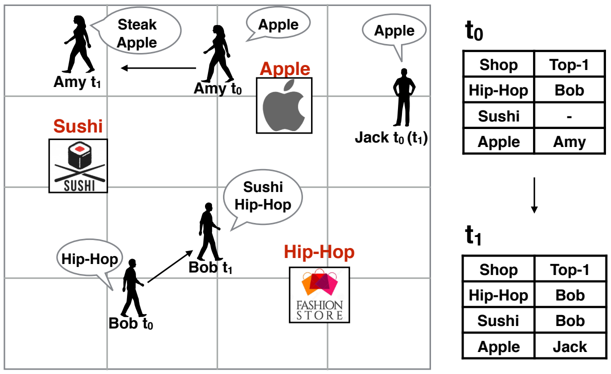
Yuyang Dong[†] Hanxiong Chen[‡] Chuan Xiao[§] Jeffrey Xu Yu[♮]
Kunihiro Takeoka[†] Masafumi Oyamada[†] Hiroyuki Kitagawa[‡]
[†]*NEC Corporation, Japan* [‡]*University of Tsukuba, Japan*
[§]*Osaka University & Nagoya University, Japan* [♮]*The Chinese University of Hong Kong, China*
{dongyuyang, k_takeoka, oyamada}@nec.com
{chx, kitagawa}@cs.tsukuba.ac.jp chuanx@ist.osaka-u.ac.jp yu@se.cuhk.edu.hk

***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 change location and interests from time to time; e.g., a user watches news, blogs, and videos while moving outside. Many online services have been developed based on continuously querying spatial-keyword objects. For instance, a real-time coupon delivery system searches for potential customers using their locations and interested keywords, and sends coupons to attract them.**

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 tracking top-k dynamic spatial-keyword objects for a given set of queries. Answering this type of queries benefit many location-aware services such as e-commerce potential customer identification, drone delivery, and self-driving stores. We develop a solution based on a grid 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. We propose a series of indexing and query processing techniques to accelerate the two procedures. We also discuss batch processing to cope with the case when multiple objects change locations and keywords in a time interval and top-k results are reported afterwards. Experiments on real and synthetic datasets demonstrate the efficiency of our method and its superiority over alternative solutions.

I. Introduction

Processing spatial-keyword data is an essential procedure in location-aware applications such as recommendation and information dissemination to mobile users. Moreover, many innovative applications in the upcoming 5G network era also rely on querying spatial-keyword data. For example, for drone delivery [3], we can monitor drones' positions and the information of packages to confirm they are working properly; for self-driving grocery stores [22], users can view the locations of stores and the goods on sale before they order. 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 change locations [15], [16], [19], [20], [28], [30], [36], [38]. Yet, real-world objects are often *dynamic* and change both locations and keywords over time. In this paper, we explore the scenario of processing 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, ranked by a scoring function of a weighted average of spatial and keyword similarities. Solving this problem benefits many underlying applications such as location-aware recommender systems and drone/self-driving store monitoring. We show a motivating example as follows.



Shop	Top-1
Hip-Hop	Bob
Sushi	-
Apple	Amy

Shop	Top-1
Hip-Hop	Bob
Sushi	Bob
Apple	Jack

Fig. 1: E-commerce coupon distribution.

Consider an e-commerce coupon distribution system in Fig. 1. A hip-hop cloth store, a sushi restaurant, and an Apple Store are registered as queries. Three users (objects) are using mobile phones. Suppose we search for top-1 results; i.e, the system identifies top-1 users as potential customers w.r.t. the three shops and sends coupons. At time t_0, Bob is watching a hip-hop music video, Amy is searching for Apple, and Jack is reading news on Apple. 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 Apple, but Amy is closer to the Apple Store 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 (we assume previous keywords do not immediately disappear in this example). Amy moves away from the Apple Store and searches for steak. So her keywords become Apple and steak. Jack is staying and still reading news on Apple. 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. Since Jack becomes closer than Amy to the Apple Store, he replaces Amy as the top-1 result of the Apple Store.

To the best of our knowledge, this is the first work targeting the case when objects change both spatial and textual attributes. For real-time applications, the key issue is to efficiently update top-k results for the queries whenever an object updates its state. Although one may convert dynamic objects to object streams by regarding an update as a deletion followed by an insertion, simply applying existing methods for related problems (e.g., CIQ [6] and SKYPE [26] for location-aware publish/subscribe systems) does not deliver sufficient efficient query processing. Our experiments show that adaptation of these methods spend more than 60 seconds to process a batch of 1,000 object updates on a dataset of 250K queries, hence difficult to keep up with the pace of frequent update in real-time applications. Next we summarize the challenges of this problem and our solution.

The first challenge originates from the large number of queries. It is prohibitive to check every query and recompute top-k results. On the other hand, the number of **affected queries** (i.e., the queries that change any top-k result) is usually very small once an object update occurs. This challenge is akin to that in location-aware publish/subscribe systems. Both CIQ and SKYPE utilize an inverted index for keywords and a quadtree for spatial information to identify the queries (referred to as subscriptions in [6], [26]) affected by an object update (referred to as messages in [6], [26]). However, CIQ and SKYPE have to traverse the quadtree and compute score upperbounds to determine whether a cell can be skipped, hence resulting in considerable cell access. In addition, both methods index single keywords in the inverted index, yet it is common that an object has only frequent keywords in our problem setting, thereby resulting in access to long postings lists in the inverted index and hence significant overhead.

To address the first challenge, we index objects and queries in a grid index and propose the notions of cell-cell links (*CC links*) and l-*signatures*. CC links model the cells in a grid as a directed graph. Whenever an object update occurs, through the outgoing CC links of the cell of the new location, we can quickly identify the cells that need look-up and avoid accessing the cells that can be pruned. l-signatures are combinations of l keywords. Despite the existence of frequent keywords, the frequencies of their combinations are often significantly smaller. By indexing l-signatures of queries and selecting a good set of l-signatures from the updated object, the cost of inverted index access can be remarkably reduced.

The second challenge is to compute the top-k results of the affected queries. Since the update of an object may cause the object to move out of the top-k results 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. The buffered objects are used when a top-k refilling is needed. Such object-based buffering is inefficient for our problem because a reordering of the objects in the buffer is required whenever an update occurs.

To address the second challenge, we first propose to leverage the upper and lower bounds of scores of the objects in a cell. The key observation is that the score bounds of the objects in a cell do not change frequently. To improve the performance, we also observe that the new top-k result of a query is either the updated object or the $(k+1)$-th result prior to the update. Thus, on top of the score bounds, we propose a method that maintains a *list of cells* for each query such that $(k+1)$-th object of the query is guaranteed to reside in one of these cells. We derive the condition of these cells and address the issue of the list update.

We extend our method to batch processing, i.e., multiple objects change state in an time interval. By taking into account of sharing computation for multiple objects, the technique can be used to handle the case when top-k results are supposed to be reported once a batch. In addition, the initialization of top-k results and the update of queries are discussed. We conduct experiments on real and synthetic datasets. The results demonstrate that the proposed techniques are effective in reducing query processing time and contribute to a substantial overall speed-up over alternative solutions, hence reducing the average query processing time to milliseconds.

Our contributions are summarized as follows. (1) 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 (Section III). (2) We propose a query processing method (Section IV) that comprises an affected query finder (Sections V) and a top-k refiller (Section VI) to address the technical challenges of the studied problem. We devise a series of data structures and algorithms for the two components. (3) We extend our method to handle the case of batch processing (Section VII). (4) We conduct extensive experiments (Section X). The results demonstrate the effectiveness of the components of our method and the superiority of our method over alternative solutions in speed.

Compared to the preliminary version of this paper [11], we made the following improvements: (1) For affected query finder, we proposed data structures (CC links and l-signatures) tailored to our problem to speed up query processing. (2) For top-k refiller, we improved the cell list maintenance policy and the list update algorithm which employs a lazy update strategy for efficiency. (3) We extended our method to batch processing. (4) We discussed initialization and query updates. (5) We conducted substantially more experiments (scalability, batch processing, etc.).

II. Related Work

Location-aware publish/subscribe systems. In location-aware publish/subscribe systems, users register their interests as continuous queries into the system, and then new streaming objects are delivered to relevant users. Keyword Boolean matching was studied in [5], [17], [27], spatial range query was studied in [8], and the case of scoring function combining spatial and keyword similarities were considered in [6], [26]. The major difference in this line of work from ours is that they do not consider dynamic objects. For our problem, although the techniques in [6], [26]

TABLE I: Comparison to related studies.

Research	Objects	Queries	Attributes
This work	dynamic	static	spatial-keyword
[6], [26]	streaming	streaming	spatial-keyword
[15], [16], [28], [38]	static	moving	spatial-keyword
[2], [7], [9], [18]	static	static	spatial-keyword
[19], [20], [30], [36]	moving	static	spatial

can be adapted to find affected queries, they cannot be used to compute top-k results for these queries.

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. Wu *et al.* [28] proposed to answer continuously moving top-k spatial-keyword (MkSK) queries using safe-regions on multiplicatively weighted Voronoi cells. Huang *et al.* [16] studied MkSK queries with a general weighted sum ranking function and proposed to use hyperbola-based safe-regions to filter objects. Zheng *et al.* [38] studied continuous Boolean top-k spatial-keyword queries in a road network. Guo *et al.* [15] 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 [10], [14] or using a scoring function [23], [37]. We refer readers to [7], [9], [18] for various problem settings and methods. The studies in this category focus on a snapshot query on static datasets, whereas our problem is continuous queries on dynamic objects. The most related work is [2], which considers spatial, keyword, and social factors for ranking. It employs a hierarchical grid index to reduce cell lookup, and processes keywords with an inverted index.

Monitoring moving objects. Another line of work [19], [20], [30], [36] aims at keeping the kNN moving objects w.r.t. a fixed query point. These solutions only consider the spatial similarity and thus cannot be directly used for our problem. For example, the objects outside the influence region [19] may be high in keyword similarity and ranked higher than those inside the influence region, rendering the filtering ineffective.

Table I compares our work to existing studies. Streaming objects/queries appear in a sliding window but do not change attributes. Moving objects/queries only change locations.

III. Preliminaries

Definition 1 (Dynamic Spatial-Keyword Object). *A dynamic spatial-keyword object o is a pair $(o.\rho, o.\psi)$. $o.\rho$ is the location of o, represented by spatial coordinates. $o.\psi$ keeps track of the 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 a triplet $(q.\rho, q.\psi, q.\alpha)$, where $q.\rho$ is a location, $q.\psi$ is a set of keywords, and $q.\alpha$ is a parameter to balance spatial and keyword similarities. The three parameters are all static.*

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

TABLE II: Frequently used notations.

Symbol	Description
o, O	an object, a set of objects
q, Q	a query, a set of queries
$SimST(o, q)$	the score of o w.r.t. q
t	a timestamp
$o^t, o^{t'}$	the state of an object o at time t/t'
$q.obj(k, t)$	the k-th object of q (ranked by score) at t
$q.obj(1\,..\,k, t)$	the set of top-k objects of q at time t
$q.score(k, t)$	the score of $q.obj(k, t)$
c, C	a cell and a grid index
$c^t, c^{t'}$	the state of a cell c at time t/t'
$o.c$, $q.c$	the cell in which o/q is located
Q_{prev}	the queries of which o is a top-k result at t
Q_{next}	the queries s.t. $SimST(o^{t'}, q) > q.score(k, t)$
τ	a keyword similarity threshold (deduced from $q.score(k, t)$)
$c^t.\psi_{\max}$, $c^t.\psi_{\min}$	the keywords in c^t, weighted by max/min in c^t
$maxscore(c^t, q)$	max. score of the objects in c to q at t
$minscore(c^t, q)$	min. score of the objects in c to q at t
$maxminscore_{<k}$	$\max\{\, minscore(c^t, q)\,\}$, $c \in C$ and $maxscore(c^t, q) < q.score(k, t)$
$q.CL$	a cell list of q at t

For brevity, we call a dynamic spatial-keyword object an *object*, and a spatial-keyword query a *query*. Our scoring function [1] is defined as follows.

Definition 3 (Spatial-Keyword Similarity). *Given an object o and a query q, their spatial-keyword similarity is*

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

For simplicity, we call $SimST(o, q)$ the *score* of o w.r.t. q, and the score of o when context is clear. It is a weighted average of spatial similarity $SimS$ and keyword (textual) similarity $SimT$. $SimS$ is calculated by the normalized Euclidean similarity:

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

where $Dist(\cdot, \cdot)$ measures the Euclidean distance and $maxDist$ is the maximum Euclidean distance in the space. $SimT$ is calculated using the keywords in $o.\psi$ and $q.\psi$:

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

where $wt(o.\psi, w)$ (or $wt(q.\psi, w)$) denotes the weight of keyword w in $o.\psi$ (or $q.\psi$). As such, $SimT$ exactly captures the cosine similarity between $o.\psi$ and $q.\psi$. We consider the

[1] This scoring function is also used in [26].

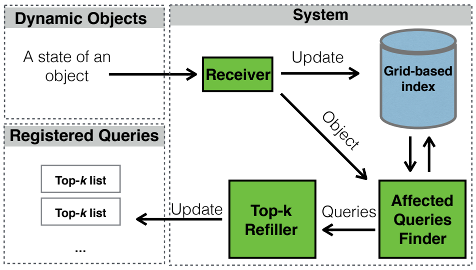


Fig. 2: Solution overview.

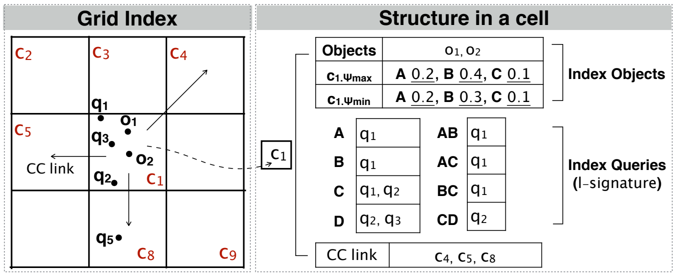


Fig. 3: Index structure.

following weighting scheme: the weights of the keywords in an object or a query are normalized to unit length, and the weight of each keyword is proportional to the keyword's inverse document frequency (idf) in a corpus. Static idf is assumed.

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

Table II lists the frequently used notations. We use superscript t/t' for the state of an object, a cell, or a grid at time t/t'.

IV. Solution Overview

We focus on in-memory solutions to the problem. Fig. 2 shows an overview of our solution. We employ a grid index, because grid index is widely used for spatial-keyword query processing and our techniques (CC links, l-signatures, etc.) favor grid index more than quadtree and R-tree. An object or a query is indexed in the cell in which it is located. We assume that in the initial state, there have already been a set of objects and a set of queries, and the top-k results of the queries have been computed. Since our focus is to solve the dynamic case of the problem, we leave the details of initialization to Section VIII. When the state of an object changes, the grid index is updated. Then we update the top-k results for 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.

V. Affected Query Finder

When a new state of an object is received, the affected query finder finds the affected queries whose top-k results or their scores need to be updated. Given o^t and $o^{t'}$, the states of an object o at two contiguous timestamps t and t', the change from o^t to $o^{t'}$ affects two sets of queries: (1) Q_{prev}: the set of queries such that o is a top-k result of q at t, and (2) Q_{next}: the set of queries such that $SimST(o^{t'}, q) > q.score(k, t)$. Q_{prev} refers to the queries to which o^t potentially moves out of top-k, while Q_{next} refers to the queries to which $o^{t'}$ is guaranteed to be a top-k result. They may overlap because o may stay as a top-k result despite the change of state. It is easy to see that the other queries in Q can be safely excluded.

To compute Q_{prev}, we assume that the top-k results of all the queries at t have been correctly computed. Hence, an *object-query map* (implemented as an inverted index) can be employed to map o to a list of queries of which o is a top-k result. The map is updated whenever a top-k result of a query changes. Computing Q_{next} is much more challenging. A sequential scan of Q is too time-consuming for real applications. We propose a method of computing Q_{next} composed of the following two procedures: (1) exploiting the grid index to find out the cells that may contain a query in Q_{next}, and (2) for each cell in (1), identifying the queries that share a necessary number of keywords with o using an efficient search algorithm. Two data structures, cell-cell link and l-signatures, are developed to handle the two procedures. Next we introduce them respectively.

A. Cell-Cell Link

Since the keyword similarity $SimT$ is no greater than 1 and the minimum distance between two cells is static, it is easy to derive a score upperbound for a query located in a different cell from $o^{t'}$'s. CIQ and SKYPE index queries in a quadtree and exploit this bound. By comparing the bound with the score of the k-th object at t, unpromising cells (i.e., those too far from $o^{t'}$ to have a bound over the k-th object's score) can be pruned by traversing the quadtree. However, when adapting CIQ or SKYPE for our problem, index access could be expensive because the pruning at the first few depths of the quadtree is less effective, meaning that we have to go deeper in the quadtree and access quite a number of cells. Moreover, even if a cell can be pruned, we only get aware of this after comparing the score bound with the smallest k-th object's score in this cell. Such comparison is invoked many times and significantly affects the performance. Seeing this inefficiency, we seek a solution to finding Q_{next} by answering the question: can we avoid the comparison so that the unpromising cells are *not even accessed*?

Given $o^{t'}.c$, the cell in which $o^{t'}$ is located, our basic idea is to store at $o^{t'}.c$ a set of links that directly point to the cells

```
Algorithm 1: AQF($o^t, o^{t'}$)
Input  : the states of $o$ at $t$ and $t'$
Output : affected queries $Q_{prev}$ and $Q_{next}$
1 $Q_{prev} \leftarrow$ the queries in the object-query map of $o$;
2 foreach $c$ connected by an outgoing CC link of $o^{t'}.c$ do
3     $Q_{next} \leftarrow Q_{next} \cup$ GetQueryIn($c, o^{t'}$);
4 return $Q_{prev}, Q_{next}$
```

having at least one possible query in Q_{next}. In doing so, we can quickly identify these cells and circumvent the access to the unpromising ones. We call these links *cell-cell links* and *CC links* for short. In this sense, the cells in the grid are the vertices of a directed graph, and there is an edge from $o^{t'}.c$ to a cell c if c potentially has a query in Q_{next}. For the sake of efficiency, the update to the CC links needs to be infrequent.

Based on this idea, we first derive the upperbound of score across cells. Given $o^{t'}.c$ and $q.c$ (the cell in which q is located), by Eq. 1, we have an upperbound of score of $o^{t'}$ and q, assuming the keyword similarity is 1:

$$UB_{SimST(o^{t'}.c,q.c)} = q.\alpha \cdot UB_{SimS(o^{t'}.c,q.c)} + 1 - q.\alpha. \quad (4)$$

$UB_{SimS(o^{t'}.c,q.c)}$ is the upperbound of spatial similarity between $o^{t'}.c$ and $q.c$: $UB_{SimS(o^{t'}.c,q.c)} = 1 - \frac{minD(o^{t'}.c,q.c)}{maxDist}$, where $minD(o^{t'}.c, q.c)$ is the minimum distance from $o^{t'}.c$ to $q.c$. By comparing $UB_{SimST(o^{t'}.c,q.c)}$ with $q.score(k, t)$, we have:

Lemma 1. *Consider an object $o^{t'}$ and a query $q \in Q_{next}$. $UB_{SimS(o^{t'}.c,q.c)} > \frac{q.score(k,t)+q.\alpha-1}{q.\alpha}$.*

Proof. By the definition of Q_{next}, $SimST(o^{t'}, q) > q.score(k, t)$. Because $UB_{SimST(o^{t'}.c,q.c)}$ is the score upperbound of $o^{t'}$ and q, $UB_{SimST(o^{t'}.c,q.c)} > q.score(k, t)$. By Eq. 4, $q.\alpha \cdot UB_{SimS(o^{t'}.c,q.c)} + 1 - q.\alpha > q.score(k, t)$. Therefore, $UB_{SimS(o^{t'}.c,q.c)} > \frac{q.score(k,t)+q.\alpha-1}{q.\alpha}$. □

The lemma means that a cell c has a query in Q_{next}, only if it is close enough to $o^{t'}.c$. For each cell c, it is obvious that only the query yielding the minimum $\frac{q.score(k,t)+q.\alpha-1}{q.\alpha}$ has to be considered. We connected the cells that satisfy the condition in Lemma 1 by CC links, each of which is an edge from $o^{t'}.c$ to c. It can be seen that only when the minimum $\frac{q.score(k,t)+q.\alpha-1}{q.\alpha}$ of a cell changes, the incoming CC links of this cell are updated. Hence we can achieve the goal of infrequent update.

Example 2. *In Fig. 3, the CC links of c_1 are displayed by solid lines. Suppose o_2 is the object that changes state. Since $o_2^{t'}$ is located in c_1, we use c_1's outgoing CC links to retrieve Q_{next}. So only c_4, c_5, and c_8 (just for illustration) are accessed.*

The pseudo-code of our affected query finder is given in Algorithm 1. Q_{prev} is obtained by the aforementioned object-query map (Line 1). For Q_{next}, we utilize the CC links from $o^{t'}.c$ to find the cells that may have a query in Q_{next} (Line 2). The cells are then processed by a function GetQueryIn (Line 3), which will be introduced next.

B. l-Signatures

Recall that in Eq. 4, the keyword similarity is assumed to be 1. Yet, we may derive a lowerbound of keyword similarity for $o^{t'}$ and any Q_{next} query in c:

Lemma 2. *Consider an object $o^{t'}$ and a query q in c. If $q \in Q_{next}$, then $SimT(o^{t'}.\psi, q.\psi) > \frac{q.score(k,t)-q.\alpha \cdot UB_{SimS(o^{t'}.c,c)}}{1-q.\alpha}$.*

Proof. By the definition of Q_{next}, $SimST(o^{t'}, q) > q.score(k, t)$. Because $UB_{SimS(o^{t'}.c,c)}$ is the upperbound of the spatial similarity between the two cells, by replacing $SimS$ in Eq. 1, we have $q.\alpha \cdot UB_{SimS(o^{t'}.c,c)} + (1 - q.\alpha) \cdot SimT(o^{t'}.\psi, q.\psi) > q.score(k, t)$, hence deducing the lemma. □

Let $\tau = \min\{ \frac{q.score(k,t)-q.\alpha \cdot UB_{SimS(o^{t'}.c,c)}}{1-q.\alpha} \mid q \in c\}$, i.e., the minimum value among all the queries in c. It is clear that if $SimT(o^{t'}.\psi, q.\psi)$ satisfies the condition in Lemma 2, then it must be greater than τ. Thus, we may use τ as a threshold, and the task of finding the Q_{next} queries in c becomes a *set similarity search* problem [4], [29].

Problem 2 (Set Similarity Search for Q_{next} Queries)**.** *Given an object $o^{t'}$ and a cell c, find the Q_{next} queries in c such that $SimT(o^{t'}.\psi, q.\psi) > \tau$.*

An immediate solution to this problem is to utilize an inverted index, mapping each keyword to a list of queries having this keyword in c, so the queries that share at least one keyword with the object can be identified as candidates to be verified. SKYPE leverages the prefix filtering technique [4], [29] that has been widely adopted for set similarity search. It sorts the keywords of the object by decreasing order of weight, and then only the first few keywords (called prefix, whose length is computed based on the weights of keywords and τ) need to be looked up in the inverted index. As we have explained in Section I, the main drawback of adapting SKYPE for our problem is that the inverted index is built on *single keywords*. The access to the inverted index causes considerable overhead as well as numerous candidates when there are only frequent keywords in $o^{t'}$ (e.g., shop and restaurant), which is common in our problem. To address this issue, we propose a signature scheme to efficiently answer the set similarity search in our problem.

Our key idea is to build an inverted index on not only single keywords but also *keyword combinations*. For example, given a query with two keywords A and B, we index A, B, and AB. We call such keyword combination (including single keywords) l-signatures, where l denotes the number of keywords in a signature. In order to control the size of a signature (as the enumeration of the l-signatures of a query is exponential in l), we set a maximum limit of l, denoted by $l_{\max}$. Given a query in a cell c, all the query's $1, \ldots, l_{\max}$-signatures are enumerated and indexed; while for $o^{t'}$, we *select* a set of l-signatures and look up in the inverted index. The queries in c having at least one of these l-signatures are regarded as candidates, which will be verified whether they are indeed Q_{next} queries.

Example 3. *Consider three queries q_1, q_2, and q_3 in Fig. 3. q_1 has keywords* `A`*,* `B`*, and* `C`*. q_2 has keywords* `C` *and* `D`*. q_3 has a keyword* `D`*. Suppose $l_{\max} = 2$. Then the l-signatures of q_1 are $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{AB}, \texttt{AC}, \texttt{BC}\}$. The l-signatures of q_2 are $\{\texttt{C}, \texttt{D}, \texttt{CD}\}$. The l-signature of q_3 is $\{\texttt{D}\}$. The corresponding inverted index is shown in Fig. 3. Suppose the selected set of l-signatures of $o^{t'}$ is $\{\texttt{A}, \texttt{CD}\}$. Then q_1 and q_2 are the candidates.*

The advantage of using l-signatures results from the observation that an l-signature ($l > 1$) is usually much less frequent than its constituent keywords; e.g., the combination of car and restaurant is much less frequent than either keyword. By looking up the postings list of the combination, the cost of index access and the candidate number can be remarkably reduced.

The signature selection of $o^{t'}$ is essential to the correctness of the algorithm and the query processing performance, since we need to guarantee that no queries satisfying $SimT(o^{t'}.\psi, q.\psi) > \tau$ will be missed and at the same time achieve a small candidate set. For this reason, we propose the notion of object variants.

Definition 4. *A variant of o, denoted by v, is a subset of $o.\psi$, the keywords of an object o, such that $SimT(o.\psi, v) > \tau$. The keyword weights of v are normalized to unit length.*

For example, $o.\psi = \{\texttt{A}, \texttt{B}, \texttt{C}\}$, and the weights are 0.332, 0.5, and 0.8, respectively. $\tau = 0.9$. Then $\{\texttt{B}, \texttt{C}\}$ is a variant since $SimT(o.\psi, \{\texttt{B}, \texttt{C}\}) = 0.94 > \tau$ (note the normalization of $\{\texttt{B}, \texttt{C}\}$). $\{\texttt{A}, \texttt{C}\}$ is not a variant since $SimT(o.\psi, \{\texttt{A}, \texttt{C}\}) = 0.87 < \tau$. Let $V(o.\psi)$ denote the set of all variants of o.

Lemma 3. *If $SimT(o.\psi, q.\psi) > \tau$, then $\exists v \in V(o.\psi)$ such that $v \subseteq q.\psi$.*

Proof. We prove by contradiction. Assume that $\nexists v \in V(o.\psi)$ such that $v \subseteq q.\psi$. Let $r_1 = o.\psi \cap q.\psi$, and the keyword weights are the same as those in $o.\psi$. Let $r_2 = r_1$, and the keyword weights are normalized to unit length. Then $SimT(o.\psi, r_2) \geq SimT(o.\psi, r_1) = SimT(o.\psi, q.\psi) > \tau$. r_2 is a variant of o. Because $r_2 = r_1$, it is a subset of $q.\psi$. This contradicts the assumption. □

The lemma means that the keywords of a Q_{next} query must be a superset of at least one variant of $o^{t'}$. We say an l-signature s *covers* a variant v, iff. the keywords of s are a subset of v; e.g., `AB` covers `ABC` but `AD` does not (we abuse the notations for keyword combinations and sets). Let $S_{all}(\cdot)$ be the set of all $1, \ldots, l_{\max}$-signatures of an object, a query, or a variant. Let $S_{sel}(\cdot)$ be a subset of $S_{all}(\cdot)$. We say that $S_{sel}(o.\psi)$ is a *valid* set of l-signatures of o, iff. $\forall v \in V(o.\psi)$, v is covered by at least one signature $s \in S_{sel}(o.\psi)$; e.g., given a $V(o.\psi) = \{\texttt{AB}, \texttt{BC}, \texttt{CDE}\}$, $\{\texttt{AB}, \texttt{C}\}$ is a valid set, since `AB` is covered by `AB`, and `BC` and `CDE` are covered by `C`. We have the following relationship between the l-signatures of a query and an object.

Lemma 4. *Consider an object o and $S_{sel}(o.\psi)$, a valid set of l-signatures of o. Given a query q, if $SimT(o.\psi, q.\psi) > \tau$, then $S_{all}(q.\psi) \cap S_{sel}(o.\psi) \neq \emptyset$.*

Proof. By Lemma 3, if $SimT(o.\psi, q.\psi) > \tau$, $\exists v \in V(o.\psi)$ such that $v \subseteq q.\psi$. Let v^* be one of such v. Because $\forall v \in V(o.\psi)$, v is covered by at least one $s \in S_{sel}(o.\psi)$, $\exists s \in S_{sel}(o.\psi)$, such that v^* is covered by s. Therefore, $s \in S_{all}(v^*)$, and thus $S_{all}(v^*) \cap S_{sel}(o.\psi) \neq \emptyset$. Because $v^* \subseteq q.\psi$, $S_{all}(v^*) \subseteq S_{all}(q.\psi)$. Therefore, $S_{all}(q.\psi) \cap S_{sel}(o.\psi) \supseteq S_{all}(v^*) \cap S_{sel}(o) \neq \emptyset$. □

By this lemma, we can select a valid set of l-signatures for $o^{t'}$. It is guaranteed that any Q_{next} query *must share* at least one of these selected signatures. For the sake of efficiency, we need to minimize the cost of the selected signatures. The lengths of postings lists in the inverted index are used here to measure the cost. Then the l-signature selection problem is defined below.

Problem 3 (l-signature Selection)**.** *Let $|L_s|$ denote the length of the postings list of a signature s in the inverted index. Given an object o and a threshold τ, the l-signature selection problem is to select $S_{sel}(o.\psi)$, a valid set of l-signatures for o, such that $\sum_{s \in S_{sel}(o.\psi)} |L_s|$ is minimized.*

It is easy to see that the l-signature selection problem is exactly a minimum weighted set cover problem. Hence it is NP-hard and can be solved by a greedy algorithm with an approximation ratio of $O(ln|V(o.\psi)|)$ [35]. Given variants $V(o.\psi)$ and all the l-signatures of o, the greedy algorithm picks signatures in the order of decreasing benefit-to-cost ratio. The benefit of an l-signature is the number of variants that it covers in the uncovered part of $V(o.\psi)$. The complexity of the greedy algorithm is $O(|S_{all}(o.\psi)| \cdot |V(o.\psi)|) = O(\binom{|o.\psi|}{l_{\max}} \cdot (2^{|o.\psi|} - 1))$. $|o.\psi|$ is usually small in real applications, and we can choose a small $l_{\max}$ to control the enumeration cost of $S_{all}(o.\psi)$. Since it is difficult to estimate the number of selected signatures by the greedy algorithm, we will choose $l_{\max}$ through empirical study rather than using a cost model.

Enumerating variants. Prior to running the greedy algorithm, a technical challenge is to compute $V(o.\psi)$. One may notice that we only need to cover the minimal variants (i.e., the variant such that we cannot remove any keyword from it and still make a variant) in $V(o.\psi)$ instead of all. However, the minimality check is costly. We take a compromise to quickly find a subset of $V(o.\psi)$ and still guarantee the selected l-signatures cover $V(o.\psi)$: First, we sort the keywords of $o.\psi$ by decreasing order of weights, and pick keywords one by one until we get a variant. It can be seen that the number of picked keywords is the minimum number of keywords of a variant. So we only enumerate subsets of $o.\psi$ whose sizes are no smaller than this number. By enumerating subsets of $o.\psi$ in increasing order of size, we can stop when reaching a size such that all the subsets of this size are variants. Let $V'(o.\psi)$ denote the set of variants we have enumerated. We replace $V(o.\psi)$ with $V'(o.\psi)$ and run the greedy algorithm. The returned l-signatures are guaranteed to cover $V(o.\psi)$, because any variant in $V(o.\psi)$ is a superset of at least one variant in $V'(o.\psi)$. For example, suppose there are 5 keywords in $o.\psi$, and the minimum number of keywords in a variant is 2. So we enumerate subsets of $o.\psi$ whose sizes are 2, 3, ... Suppose all the subsets of size 3 are variants. We stop at 3 and $V'(o.\psi)$ includes only variants of size 2 and 3.

Algorithm 2: GetQueryIn$(c, o^{t'})$

```
1 $\tau \leftarrow \min\{ \frac{q.score(k,t) - q.\alpha \cdot UB_{SimS(o^{t'}.c,c)}}{1-q.\alpha} \mid q \in c \}$;
2 $V' \leftarrow$ GenerateVariants$(o^{t'}, \tau)$;
3 $S_{all} \leftarrow$ all $l$-signatures of $o^{t'}$;
4 $S_{sel} \leftarrow$ SelectSignatures$(V', S_{all})$;
5 $Q_{next} = \emptyset$;
6 foreach $s \in S_{sel}$ do
7     foreach $q \in c.L_s$ do
8         if $SimST(o^{t'}, q) > q.score(k,t)$ then
              $Q_{next} \leftarrow Q_{next} \cup \{ q \}$;
9 return $Q_{next}$
```

Algorithm 3: SelectSignatures(V, S_{all})

```
1 $S_{sel} \leftarrow \emptyset$;
2 foreach $s \in S_{all}$ do
3     $cost_s \leftarrow |L_s|$;
4 while $V \neq \emptyset$ do
5     $cover_s \leftarrow$ the set of variants in $V$ covered by $s$;
6     $benefit_s \leftarrow |cover_s|$;
7     $S_{sel} \leftarrow S_{sel} \cup \{s \mid s \in S_{all} \wedge s$ has maximum
        $benefit_s/cost_s\}$;
8     $S_{all} \leftarrow S_{all} \backslash \{ s \}, V \leftarrow V \backslash \{ cover_s \}$;
9 return $S_{sel}$
```

By putting it all together, the pseudo-code of the processing of each cell is provided in Algorithm 2. It first computes the τ threshold of the cell (Line 1). The V' variants of $o^{t'}$ are generated (Line 2). We enumerate the l-signatures of $o^{t'}$ (Line 3) and select the ones that cover V' (Line 4). For each selected signature, we access the corresponding postings list of the inverted index of this cell (Line 7). Each query in the list is a candidate and inserted into Q_{next} if the score between $o^{t'}$ and the query exceeds the k-th result at t (Line 8). The pseudo-code of the signature selection algorithm (SelectSignatures in Line 4) is given in Algorithm 3.

One signature selection for all. When invoking Algorithm 2 in Algorithm 1, a limitation is that signature selection is invoked for each cell connected by a CC link and thus causes redundancy. We observe that the variant set $V(o.\psi)$ only depends on the object and the threshold τ. Given two cells c_1 and c_2 with thresholds τ_1 and τ_2, respectively, it can be seen that the corresponding variant sets $V_1(o.\psi) \subseteq V_2(o.\psi)$ if $\tau_1 \geq \tau_2$. This suggests the l-signatures selected using $V_2(o.\psi)$ can cover $V_1(o.\psi)$ as well. Hence, we may take the minimum τ across all the cells identified by CC links, and run the greedy algorithm only *once* to obtain the l-signatures for all these cells. First, we generate the variant set $V'(o.\psi)$ by the minimum τ. While running the greedy algorithm, because $V_1(o.\psi) \subseteq V_2(o.\psi)$, $V_1(o.\psi)$ is always covered prior to $V_2(o.\psi)$. So we monitor for each cell the time when the corresponding variant set has just been covered, and record the l-signatures selected so far as the l-signatures for this cell.

We briefly discuss the differences of l-signatures from existing studies, since keyword (token) combinations were also used for approximate set containment search [1] and local similarity search [25]: (1) [1] enumerates keyword combinations frequency, while we use number of keywords to control the enumeration. (2) [1] covers minimal variants, while we choose a compromise between minimal and all. (3) [25] is based on prefix filtering and partitioning. It is not applicable for low thresholds since the prefix length may exceed the size of an object. Our method does not have such limitation. (4) We propose the one-selection-for-all technique tailored to our problem setting.

VI. Top-k Refiller

After identifying Q_{prev} and Q_{next} by the AQF, we update the top-k results for the queries in the two sets. For those in Q_{next}, the update is straightforward because we only need to insert $o^{t'}$ into the query's top-k and delete the previous k-th object. For those in Q_{prev}, since $o^{t'}$ may move out of the top-k, we need to refill the top-k with another object. A sequential scan of O is prohibitively expensive. One option is to borrow the top-k refilling method in kmax [34]. It maintains top-k' results where k' is a value between k and a maximum buffer size $k_{\max}$. As discussed in Section I, keeping objects in a buffer is inefficient in our problem setting because objects are dynamic, incurring frequent buffer maintenance and hence considerable overhead. Next we propose our method to deal with top-k refilling.

A. Deriving Score Bounds of Cells

We first explore in the direction of exploiting the score bounds of cells. It has the following advantages: (1) the number of cells is much smaller than the number of objects (compared to the buffering strategy in kmax); (2) the bound of spatial similarity from a cell to a query is static; and (3) despite objects being dynamic, the bounds of keyword similarities from the objects in a cell to a query do not change frequently.

The bound of spatial similarity from a cell to a query is easy to derive. To bound keyword similarities, additional information needs to be stored in the grid index. For each cell c, we collect the distinct keywords that appears in the objects in c at t, denoted by $c^t.\psi$. Then we have two weights for each keyword, the maximum and the minimum weights of w among all the objects in c that have w at t. For simplicity, we use $c^t.\psi_{\max}$ and $c^t.\psi_{\min}$ to distinguish the weight we want to use, though they refer to the same set of keywords $c^t.\psi$. When an update of an object occurs, such information is updated accordingly.

Example 4. *In Fig. 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: $\texttt{A}$, $\texttt{B}$, and $\texttt{C}$. Then $c_1^t.\psi = \{ \texttt{A}, \texttt{B}, \texttt{C} \}$. The weights in $c_1^t.\psi_{\max}$ are $\{ 0.2, 0.4, 0.1 \}$. The weights in $c_1^t.\psi_{\min}$ are $\{ 0.2, 0.3, 0.1 \}$.*

With $c^t.\psi_{\max}$ and $c^t.\psi_{\min}$, the upper and lower bounds of keyword similarity in c are bounded, as computed by Eq. 3. Then we bound the scores of the objects in a cell.

$$maxscore(c^t, q) = q.\alpha \cdot UB_{SimS(c,q)} + (1 - q.\alpha) \cdot UB_{SimT(c^t,q)}.$$
$$minscore(c^t, q) = q.\alpha \cdot LB_{SimS(c,q)} + (1 - q.\alpha) \cdot LB_{SimT(c^t,q)}.$$

UB_{SimS} and LB_{SimS} are upper and lower bounds of spatial similarity, respectively: $UB_{SimS(c,q)} = 1 - \frac{minD(c,q.\rho)}{maxDist}$, $LB_{SimS(c,q)} = 1 - \frac{maxD(c,q.\rho)}{maxDist}$, where $minD(c,q.\rho)$ and

$maxD(c, q.\rho)$ denote the minimum and the maximum distances from a cell to the query location, respectively. UB_{SimT} and LB_{SimT} are upper and lower bounds of keyword similarity, respectively: $UB_{SimT(c^t,q)} = SimT(c^t.\psi_{\max}, q.\psi)$ and $LB_{SimT(c^t,q)} = \min \{ wt(c^t.\psi_{\min}, w) \cdot wt(q.\psi, w) \mid w \in c^t.\psi_{\min} \cap q.\psi \}$. Note that these bounds are tight bounds because objects may reside on cell boundaries and contain the same keywords (and weights) as $c^t.\psi_{\max}$ or $c^t.\psi_{\min}$.

B. Maintaining Cell Lists

One may design an algorithm to leverage the derived score bounds for pruning, yet a caveat is that a sequentially scan of all the cells in the grid should be avoided. To this end, we notice that when o^t changes to $o^{t'}$, at most one object in the top-k of q changes. Moreover, we have the following observation.

Observation 1. *Consider an old state o^t and a new state $o^{t'}$ of an object. For any $q \in Q_{prev}$, $q.o(1\mathinner{..}k, t') \setminus q.obj(1\mathinner{..}k, t) = \{ o \}$ or $\{ q.obj(k+1, t) \}$.*

$q.obj(k+1, t)$ is the $(k+1)$-th result of q at time t. The observation indicates that we only need to compare $o^{t'}$ and $q.obj(k+1, t)$, and pick the one with the higher score as the new top-k result of q. Based on this observation, we are able to exploit the score bounds in a more effective way by maintaining a cell list (CL) for each q such that the $(k+1)$-th object of q at t is guaranteed to reside in one of these cells [2]. We have:

Lemma 5. *If $minscore(c^t, q) > q.score(k, t)$, then $q.obj(k+1, t) \notin c^t$.*

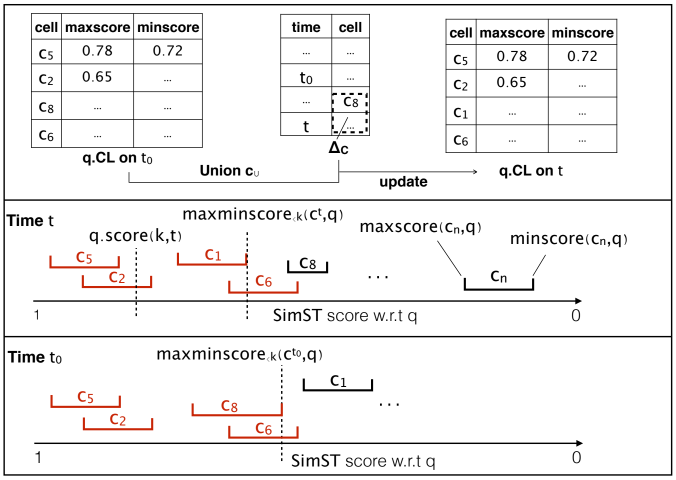
Proof. Because $minscore(c^t, q) > q.score(k, t)$, we have $\forall o_i \in c^t$, $SimST(o_i^t, q) > q.score(k, t) \geq q.score(k+1, t)$. Therefore, $q.obj(k+1, t) \notin c^t$. □

Let $maxminscore_{<k}(C^t, q) = \max\{ minscore(c^t, q) \mid c \in C \wedge maxscore(c^t, q) < q.score(k, t) \}$; i.e., we collect the cells whose $maxscore$ values are less than $q.score(k, t)$, and pick the the maximum of their $minscore$ values.

Lemma 6. *If $maxscore(c^t, q) < maxminscore_{<k}(C^t, q)$, then $q.obj(k+1, t) \notin c^t$.*

Proof. We prove by contradiction: assume $q.obj(k+1, t) \in c^t$. Let c^* be the cell that yields $maxminscore_{<k}(C^t, q)$; i.e., $maxminscore_{<k}(C^t, q) = minscore(c^{*^t}, q)$. $maxscore(c^t, q) < maxminscore_{<k}(C^t, q)$, so $maxscore(c^t, q) < minscore(c^{*^t}, q)$. Because $maxscore$ and $minscore$ are scores' upper and lower bounds, respectively, and $q.obj(k+1, t) \in c^t$, we have $\exists o' \in c^{*^t}$, s.t. $SimST(o'^t, q) > q.score(k+1, t)$. By the definition of $maxminscore_{<k}$, $maxscore(c^{*^t}, q) < q.score(k, t)$. Therefore, $SimST(o'^t, q) < q.score(k, t)$, meaning that $SimST(o'^t, q) \leq q.score(k+1, t)$. This contradicts $SimST(o'^t, q) > q.score(k+1, t)$. □

Intuitively, the two lemmata state that if a cell whose score bounds are too high or too low, then the $(k+1)$-th result is not in it. So by taking the complement of these cells in the grid, we are guaranteed to include the $(k+1)$-th result's cell:



cell	maxscore	minscore
c5	0.78	0.72
c2	0.65	...
c8	...	...
c6	...	...

q.CL on t0

time	cell
...	...
t0	...
...	c8
t	...

cell	maxscore	minscore
c5	0.78	0.72
c2	0.65	...
c1	...	...
c6	...	...

q.CL on t

Fig. 4: top-k refilling.

Corollary 1. *Consider a set of cells $C' = \{ c \in C \mid minscore(c^t, q) \leq q.score(k, t) \wedge maxscore(c^t, q) \geq maxminscore_{<k}(C^t, q) \}$. $\exists c \in C'$, s.t., $q.obj(k+1, t) \in c^t$.*

Proof. We prove by contradiction. If $\nexists c \in C'$, s.t., $q.obj(k+1, t) \in c^t$, by the definition of C', the cell that has $q.obj(k+1, t)$, denoted by c', satisfies either $minscore(c'^t, q) > q.score(k, t)$ or $maxscore(c'^t, q) < maxminscore_{<k}(C^t, q)$. This either contradicts Lemma 5 or Lemma 6. □

Although it is sufficient to fetch the $(k+1)$-th result with only C' in Corollary 1, we also store the cells having the top-$1, \ldots, k$ results. This is to simplify the update of CL (introduced later). As such, we define the $(k+1)$-CL of q:

Definition 5 (($k+1$)-CL)**.** *The $(k+1)$-CL of a query q with a timestamp t is defined as $\{c \in C \mid maxscore(c^t, q) \geq maxminscore_{<k}(C^t, q)\}$.*

Compared to Corollary 1, we remove the condition on $minscore$. In the rest of this section, we mean $(k+1)$-CL when CL is mentioned, and we use $q.CL$ to denote the CL of q. Since we will use a lazy update strategy, the corresponding timestamp is recorded along with the CL, denoted by $q.CL.t$.

Example 5. *In Fig. 4 (time t), cells are represented by intervals. The two ends indicate the $maxscore(c^t, q)$ and $minscore(c^t, q)$ values of a cell. By the definition of $(k+1)$-CL, $q.CL$ with timestamp t is $\{ c_5, c_2, c_1, c_6 \}$.*

We design a top-k refilling algorithm utilizing CL (Algorithm 4. For any query in Q_{next}, we only insert $o^{t'}$ into the query' top-k results (Line 2). Here, we assume that $q.obj(1\mathinner{..}k, t)$ is a priority queue of size k, whose elements are ranked by the score. The $push()$ function inserts an object to the priority queue and only keeps the top-k ones. For any query in $Q_{prev} \backslash Q_{next}$ (the set difference is because the overlapping queries have been

[2] One may want to keep only the $(k+1)$-th object, but this object may change state while q is outside both Q_{prev} and Q_{next}, hence difficult to track.

Algorithm 4: TopKRefiller-CL($o^t, o^{t'}, OutQ, InQ$)

```
   Input  : the states of $o$ at $t$ and $t'$, $Q_{prev}$, $Q_{next}$
   Output : top-$k$ results for $q \in Q_{prev} \cup Q_{next}$
1  for each $q \in Q_{next}$ do
2      $q.obj(1\,..\,k, t') \leftarrow q.obj(1\,..\,k, t).push(o)$;
3  foreach $q \in \{Q_{prev} \backslash Q_{next}\}$ do
4      $q.obj(1\,..\,k, t') \leftarrow q.obj(1\,..\,k, t).pop(o)$;
5      UpdateCL($q.CL, q, t$);
6      $q.CL_{\le k} \leftarrow \{\, c \mid c \in q.CL \wedge minscore(c^t, q) \le q.score(k, t) \,\}$;
7      Sort $q.CL_{\le k}$ by descending $maxscore(c^t, q)$;
8      $score_{k+1} \leftarrow -\infty$;
9      foreach $c \in q.CL_{\le k}$ do
10         if $maxscore(c^t, q) \le score_{k+1}$ then break;
11         foreach $o' \in c$ such that $o' \neq q.obj(k, t)$ do
12             if $q.score(k, t) \ge SimST(o'^t, q) > score_{k+1}$ then
13                 $o_{k+1} \leftarrow o'$, $score_{k+1} \leftarrow SimST(o'^t, q)$;
14     if $SimST(o^{t'}, q) \ge score_{k+1}$ then $q.obj(1\,..\,k, t').push(o)$;
15     else $q.obj(1\,..\,k, t').push(o_{k+1})$;
16 return $\{\, q.obj(1\,..\,k, t') \mid q \in Q_{prev} \cup Q_{next} \,\}$
```

processed in Line 2), we remove o from the top-k (Line 4) [3] and update the query's CL to the state at t (not t', Line 5). Then we use the part of the CL that meet the condition in Corollary 1 (Line 6) to find the $(k+1)$-th result. The cells in this part are sorted by descending order of $maxscore$ (Line 7). We iterate through the cells in the partial CL. The $(k+1)$-th result at t is kept in a temporary object o_{k+1} along with its score $score_{k+1}$ during the iteratoin. If the $maxscore$ of a cell is no better than the temporary object's score (Line 10), the iteration is early terminated because unseen cells are even lower in $maxscore$. Finally, we compare o_{k+1} with o and insert the one with the higher score to the top-k results of q (Lines 14 – 15).

Example 6. *We consider a query q in $Q_{prev} \backslash Q_{next}$. In Fig. 4, the cell list of q with timestamp t includes c_5, c_2, c_1, and c_6. Since c_5 violates the condition in Corollary 1 ($minscore(c^t, q) \le q.score(k, t)$), we only access c_2, c_1, and c_6 to retrieve the $(k+1)$-th result at t. Then it is compared with $o^{t'}$. The one with the higher score is refilled to the top-k of q at t'.*

C. Cell List Update

An important step in Algorithm 4 is the update of CL (Line 5), as we need to guarantee the CL has all the cells satisfying the condition in Definition 5. A naive way of update is to scan all the cells to make a CL from scratch, but this essentially reduces the algorithm to a sequential scan of all the cells. An efficient way is to find the cells to be inserted or deleted from the existing CL. The challenge is that the CL's timestamp (say, t_0) prior to the update could be very old compared to t. Nonetheless, by Definition 5, the cells in the CL are determined by $maxscore$, $minscore$, and $q.score(k, t)$. $maxscore$ and $minscore$ depend only on the keywords that appear in the cell. We may keep track of the cells whose $c.\psi_{\max}$ or $c.\psi_{\min}$ change, along with the timestamp at which the change happens. The new CL can be computed using the old CL and the set of the cells that change $c.\psi_{\max}$ or $c.\psi_{\min}$ during time interval $(t_0, t]$. Let Δ_C denote this set. We describe our solution to CL update.

The key idea is to compute $maxminscore_{<k}(C^t, q)$, so the inserted/deleted cells can be determined by comparing their $maxscore$ values with $maxminscore_{<k}(C^t, q)$. To this end, we derive the following lemma.

Lemma 7. *Consider $q.CL$, a cell list to be updated, whose timestamp is t_0. Let $C_\cup = q.CL \cup \Delta_C$. If $maxminscore_{<k}(C^t_\cup, q) \ge maxminscore_{<k}(C^{t_0}, q)$, then we have $maxminscore_{<k}(C^t, q) = maxmins(C^t_\cup, q)$.*

Proof. By the definition of $maxminscore_{<k}$, for any $C' \subseteq C$, $maxminscore_{<k}(C^t, q) \ge maxminscore_{<k}(C'^t, q)$. Because $C_\cup \subseteq C$, $maxminscore_{<k}(C^t, q) \ge maxminscore_{<k}(C^t_\cup, q)$. Therefore, if $maxminscore_{<k}(C^t, q) \neq maxminscore_{<k}(C^t_\cup, q)$, $maxminscore_{<k}(C^t, q) > maxminscore_{<k}(C^t_\cup, q)$. We assume this is true and prove by contradiction. In this case, $\exists c' \in C \backslash C_\cup$, s.t., $maxscore(c'^t, q) > maxminscore_{<k}(C^t_\cup, q)$. Because $maxminscore_{<k}(C^t_\cup, q) \ge maxminscore_{<k}(C^{t_0}, q)$, and any cell in $C \backslash C_\cup$ does not change $maxscore$ or $minscore$ during $(t_0, t]$, we have $maxscore(c'^t, q) = maxscore(c'^{t_0}, q) > maxminscore_{<k}(C^{t_0}, q)$. This means $c' \in q.CL$ and thus $c' \in C_\cup$, which contradicts $c' \in C \backslash C_\cup$. □

The lemma reveals under which condition we can make the new CL using the old CL and Δ_C. First, we update the $maxscore$ and $minscore$ values of the cells in $q.CL$ and Δ_C for timestamp t, and compute $maxminscore_{<k}(C^t_\cup, q)$ using these $maxscore$ and $minscore$ values and $q.score(k, t)$. Then we check if $maxminscore_{<k}(C^t_\cup, q)$ is no less than that of the old CL, i.e., $maxminscore_{<k}(C^{t_0}, q)$. If so, $maxminscore_{<k}(C^t, q)$ is exactly $maxminscore_{<k}(C^t_\cup, q)$. All the cells to be inserted or deleted must belong to $C_\cup$. This is because the cells having $maxscore(c^t, q) \ge maxminscore_{<k}(C^{t_0}, q)$ are a superset of those having $maxscore(c^t, q) \ge maxminscore_{<k}(C^t_\cup, q) = maxminscore_{<k}(C^t, q)$, and the former are stored in $C_\cup$. Thus, the new CL can be made using only $C_\cup$. Otherwise (i.e., when $maxminscore_{<k}(C^t_\cup, q) < maxminscore_{<k}(C^{t_0}, q)$), we scan all the cells of the grid to make the new CL.

The pseudo-code of the above process is captured by Algorithm 5. Lines 1 – 4 update $maxscore$ and $minscore$ in $C_\cup$ and compute $maxminscore_{<k}(C^t_\cup, q)$. Lines 5 – 9 make the new CL using the cells in $C_\cup$ only. Lines 11 – 13 make the new CL by scanning all the cells in the grid. The timestamp of the CL is updated to t eventually (Line 14).

Example 7. *As shown in Fig. 4, we assume that the old $q.CL$ with timestamp t_0 is $\{\, c_5, c_2, c_8, c_6 \,\}$. $maxminscore_{<k}(C^{t_0}, q)$ is given by $minscore(c_6^{t_0}, q)$. Suppose $\Delta_C = \{\, c_1, c_8 \,\}$. So $C_\cup = \{\, c_5, c_2, c_8, c_6, c_1 \,\}$. $maxminscore_{<k}(C^t_\cup, q)$ is given by $minscore(c_1^t, q)$.*

[3] A special case is $SimST(o^{t'}, q) = q.score(k, t)$. o is the k-th result at t' and no further action is required. We omit it in the pseudo-code for conciseness.

Algorithm 5: UpdateCL($q.CL, q, t$)

```
$\Delta_C \leftarrow$ the cells that change $c.\psi_{\max}$ or $c.\psi_{\min}$ in $(q.CL.t, t]$;
foreach $c \in \Delta_C$ do
    Compute $maxscore(c^t, q)$ and $minscore(c^t, q)$;
$C_\cup \leftarrow q.CL \cup \Delta_C$, $\gamma \leftarrow maxminscore_{<k}(C^t_\cup, q)$;
if $\gamma \geq maxminscore_{<k}(C^{q.CL.t}, q)$ then
    foreach $c \in q.CL$ do
        if $maxscore(c^t, q) < \gamma$ then $q.CL.delete(c)$;
    foreach $c \in \Delta_C \backslash q.CL$ do
        if $maxscore(c^t, q) \geq \gamma$ then $q.CL.insert(c)$;
else
    $q.CL.clear()$, $\gamma \leftarrow maxminscore_{<k}(C^t, q)$;
    foreach $c \in C$ do
        if $maxscore(c^t, q) \geq \gamma$ then $q.CL.insert(c)$;
$q.CL.t \leftarrow t$;
```

Since it is greater than $maxminscore_{<k}(C^{t_0}, q)$, $maxminscore_{<k}(C^t, q) = maxminscore_{<k}(C^t_\cup, q)$. So only the cells in $C_\cup$ are considered to compute the new CL. We compare their $maxscore(c^t, q)$ values with $maxminscore_{<k}(C^t, q)$. c_8 is deleted and c_1 is inserted. The new CL is $\{\, c_5, c_2, c_1, c_6 \,\}$.

VII. Batch Processing

To extend to batch processing, we assume that multiple objects change states during time interval $(t, t']$. Let $O^{t,t'}$ denote this set of objects. Given the top-k results of each query at t, our task is to compute the top-k results at t'.

A. Affected Query Finder for Batch

We first modify the definitions of Q_{prev} and Q_{next}: (1) Q_{prev}: the multiset of queries such that $\exists o \in O^{t,t'}$, o is a top-k result of q at t, and (2) Q_{next}: the multiset of queries such that $\exists o \in O^{t,t'}$, $SimST(o^{t'}, q) > q.score(k, t)$. The occurrence of a query q in Q_{prev}/Q_{next} is the number of objects in $\in O^{t,t'}$ satisfying the above condition. The top-k results are not affected if a query is in neither Q_{prev} nor Q_{next}.

The computation of Q_{prev} is straightforward. We scan the objects in $O^{t,t'}$ and fetch Q_{prev} queries by the object-query map. To find Q_{next} queries, we scan the objects in $O^{t,t'}$ and use CC links to identify the cells that need look-up. An important observation is that objects may share keywords and computation can be shared when invoking Algorithm 2. Consider a cell c identified by CC links from multiple objects in $O^{t,t'}$. Each object o yields its own keyword similarity threshold. We generate the l-signature set S_{all} and the variant set V' for each object. Since these l-signature sets and variant sets may overlap, we take them together as the input of the signature selection algorithm. While running the signature selection, we monitor the time when the variant set of an object has been covered, and record the l-signatures selected so far. This resembles the one-signature-selection-for-all technique described in Section V-B. Besides, the two techniques can work together. As such, only one run of signature selection outputs the l-signatures for all the $O^{t,t'}$ objects and all the cells identified by CC links.

B. Top-k Refiller for Batch

Since Q_{prev} and Q_{next} are multisets and may overlap, for each query q, we count its occurrences in Q_{prev} and Q_{next} to determine what objects need to be refilled to top-k. Let Δ_q denote the difference of q's occurrences in Q_{prev} and Q_{next}. If $\Delta_q \leq 0$, the processing is similar to Q_{prev} queries in the case of single object update: we pop out the Q_{prev} objects (i.e., the $O^{t,t'}$ objects which are q's top-k results at t), and push into q's top-k the Q_{next} objects (i.e., the $O^{t,t'}$ objects such that $SimST(o^{t'}, q) > q.score(k, t)$). If $\Delta_q > 0$, the processing is similar to Q_{next} queries in the case of single object update. The Q_{prev} objects are popped out first. The CL of q has to include the $2k$-th object at t, because Δ_q can be up to k. To this end, $maxminscore_{<k}(C^t, q)$ is replaced by $kminscore_{<k}(C^t, q)$, which is k-th largest $minscore(c^t, q)$ of the cells in C such that $maxscore(c^t, q) < q.score(k, t)$. Definition 5 is replaced by $(2k)$-CL. The related lemmata, corollary, and algorithms are modified accordingly. Then we use the $(2k)$-CL to fetch the $(k+1)$-th to $2k$-th objects of q at t. They are compared with the Q_{prev} objects and the better ones are pushed into q's top-k, along with the Q_{next} objects.

VIII. Initialization and Query Updates

Initialization. We first create a grid index on the objects. The initialization of top-k results for each query is a snapshot spatial-keyword search. We do not adopt existing methods (surveyed in Section II) as they either target different problem settings or use different indexes. Our method maintains a list of temporary top-k results (initialized as empty). By starting from the query's cell, it iterates cells by a breadth-first search. The objects in each cell are checked whether they outperform any temporary top-k result. This is repeated until we reach the cells that are too far to make a score better than the current k-th one. After obtaining top-k results, the query is indexed in the grid. We create an object-query map for each object and a cell list for each query. Then we create the inverted index and CC links for each cell, and compute the keyword similarity threshold τ.

Query insertion. Like initialization, we also use snapshot spatial-keyword search to retrieve the top-k results of the inserted query. The difference is that the insertion is an online operation, so we need to optimize the processing speed. For the above snapshot spatial-keyword search method, a good set of initial top-k results help prune unpromising cells and objects. An observation is that the top-k results of two similar queries tend to resemble. For this reason, we first find the nearest neighbor query of the inserted query, and use the nearest neighbor's top-k results as an initial set of top-k for the inserted query. Then we run the snapshot spatial-keyword search. Finally, the cell list of the inserted query is created and corresponding data structures (grid, object-query map, etc.) are updated.

Query deletion. The query and its cell list are deleted. The grid, the object-query map, and the inverted index of the query's cell are updated. If the query is the one that determines the CC links or τ of its cell, the CC links or τ needs recomputation.

To efficiently process the update inverted index, we directly append an inserted query to the postings lists in the inverted index. When a query is deleted, we do not delete it from the postings lists but mark it as deleted so it will not be returned as a candidate. A postings list is reconstructed only if half of its entries are marked as deleted. Besides, we do not dynamically tune $l_{\max}$ but use a fixed value because (1) dynamically changing $l_{\max}$ may affect other queries that have already been index; and (2) our experiments show that the best $l_{\max}$ value is in the range of $[2, 4]$ across different datasets and update frequency, and the performances with different $l_{\max}$ values in this range are close.

IX. Extension

In our problem setting, we rank results by spatial-keyword similarity. Since real-world systems are typically expected to continuously monitor a variety of queries, we discuss the extension of our method to several common query types:

- **Top-k queries**: Our scoring function (Equation 1) is defined in a top-k fashion combining spatial and keyword similarities. By setting $\alpha = 0$ (or 1, respectively), the problem becomes exactly finding top-k results ranked by keyword (or spatial, respectively) similarity. Our method can support this case without any modification.
- **Spatial keyword range queries**: We suppose that results must locate in the range specified by the query and are ranked by keyword similarity. In this case, we first employ the grid index to look up the cells that overlap a query's range, and then find affected queries and refill top-k results using our method. To utilize l-signatures, we remove spatial similarity (UB_{SimS}) from the threshold τ (Section V-B). To utilize the cell list method for top-k refiller, we remove spatial similarity from the definitions of $minscore$ and $maxscore$ (Section VI-A).
- **Boolean kNN queries**: To support the Boolean semantics, we modify the inverted index as follows: a query is only indexed in the postings list of its maximal l-signatures (e.g., if a query has three keywords A, B, and C, and $l_{\max} = 2$, then it is only indexed in the postings lists of AB, AC, and BC, but not A, B, or C). To process queries, we first utilize CC links to look up the cells that may contain an affected query, and find affected queries by looking up the postings lists of the l-signatures contained in the updated object. Then we use the cell list method to refill top-k results. Keyword similarity (UB_{SimT}) is removed from the definitions of $minscore$ and $maxscore$.
- **Boolean range queries**: We use the same inverted index as for Boolean kNN queries. To process queries, we first employ the grid index to look up the cells that overlap a query's range, and find affected queries by looking up the postings lists of the l-signatures contained in the updated object. Since there is no ranking for objects, the updated object is directly inserted or deleted from the affected query's result list.

X. Experiments

A. Setting

Datasets. We used two real datasets and one synthetic dataset. Table III shows the statistics.

TABLE III: Datasets statistics.

Datasets	YELP	TWITTER	SYN
Data size	1.2M	20M	50M
Default # of objects	220K	3.5M	10M
Default # of queries	192K	3.5M	10M
Total # of keywords	819K	5.4M	6M
# of kw. per object/query	5.9	6.7	3

- **YELP** is a public dataset with over 192K businesses' information and 1.2M reviews from the Yelp business directory service [33]. We extracted the businesses' locations and descriptions as queries. Users were regarded as objects. For each user, we sort the user's reviews by timestamp order. Then for each review, we remove stop words and take the rest as keywords. So the keywords change with timestamp. Since locations are not covered by the reviews, we paired each review with the locations in a real taxi trip data [21].
- **TWITTER** is a dataset with 20M geotagged tweets from 7.1M users [24]. We randomly selected a subset of users as queries, using their locations and keywords chosen from their first tweets: for each tweet, we first randomly pick a number n in $[1, 5]$, and then take the top-n nouns ranked by descending order of inverse document frequency. Then we randomly selected a subset of users from the rest of the dataset as objects, locations and keywords extracted in the same way as queries (the only difference is that for objects, we use all the user's tweets and sort by timestamp order).
- **SYN** is a synthetic data containing 50M spatial keyword tuples. We used a dataset of moving points generated by the BerlinMOD benchmark [12] for locations, and randomly generated keywords by a Zipfian distribution.

Algorithms. We consider the following methods for affected query finder (AQF): (1) GSKI, the method in [2] based on hierarchical grid and inverted index; (2) CIQ, the method in [6] based on quadtree and inverted index, (3) SKYPE, the method in [26] based on quadtree and prefix filtering; (4) IC, the method in our previous work [11] based on grid; and (5) CCLS, our method that utilizes **CC** links and **l-s**ignatures. We consider the following methods for top-k refiller (TR): (1) kmax, the method with $kmax$ buffer in [34]; (2) PCL, the method in our previous work [11] based on cell list; and (3) CL, our cell list method in this paper. We use "A-B" to denote the combination of methods A and B, e.g., CIQ-kmax.

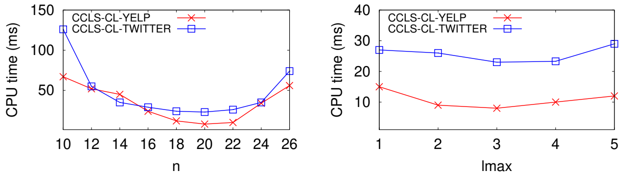
Environments and measures. The experiments were run on a MacBook Pro with a 2.2GHz Intel Core i7 CPU and 32GB memory. All the algorithms were implemented in C++ and in a main memory fashion. We randomly monitored 10,000 updates of dynamic objects and reported the average processing time per object update. The default k is 20. The default $q.\alpha$ of each query is a random value in $[0, 1]$ by uniform distribution.

B. Parameter Tuning

There are two parameters in our methods: the grid size and $l_{\max}$, the maximum number of keywords in an l-signature. We use an $n \times n$ grid here. Fig. 5a shows how the running times of

TABLE IV: Best parameter choice (SYN).

# of kw. per object/query	1	2	3	4	5
best n	18	20	20	20	20
best $l_{\max}$	1	2	3	3	3



(a) Varying grid size.

(b) Varying $l_{\max}$.

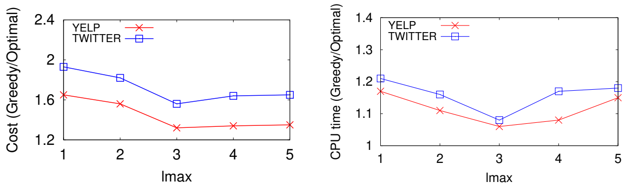
Fig. 5: Parameter tuning.

our methods vary with n. The grid size mainly affects CC links and the pruning power in TR. The best choice of n is around 16 to 20. Too few cells reduce the pruning power, while too many cells result in more cell access. Fig. 5b shows how $l_{\max}$ affects the overall performance. When $l_{\max} = 1$, our method is the same as indexing single keywords. It can be seen that using keyword combinations is better than single keywords. The best choice is $l_{\max} = 3$. When we use too large $l_{\max}$, the performance drops due to more enumeration cost. We choose $n = 20$ and $l_{\max} = 3$ in the rest of the experiments.

Table IV shows the best (fastest) n and $l_{\max}$ settings on the synthetic dataset SYN with varying number of keywords per object/query. The best n centers is almost constant (20). The best $l_{\max}$ equals to the number of average number of keywords when the latter is no more than 3, and then keeps 3 when we have more keywords in an object/query. This is expected, because there is no need to combine more keywords when the average number of keywords is small (1 – 3). When there are more keywords in an object/query, the best $l_{\max}$ does not increase due to the enumeration cost.

C. Signature Selection

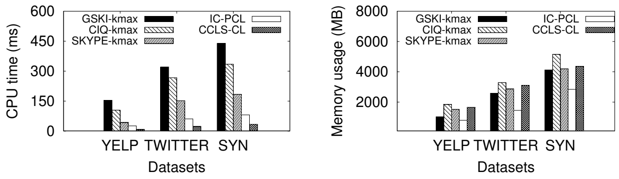
To evaluate the performance of the greedy signature selection algorithm, we vary $l_{\max}$ and plot in Fig. 6a the ratio of two costs: (1) the cost of the signatures selected by the greedy algorithm, and (2) the optimal cost (denoted as Optimal). The ratio ranges from 1.3 to 1.6 on YELP and 1.5 to 1.9 on TWITTER, showcasing that our signature selection produces a cost close to the optimal value. Note that computing the optimal cost is NP-hard and render the overall query processing much slower, on average by 4.3 times on YELP and 5.1 times on TWITTER. We also study what if optimal signatures are selected. We compare the average query processing time of our method with the optimal time, supposing that the greedy algorithm selected the signatures that produce the optimal cost. Fig. 6b shows their ratio. The ratio is slightly smaller than the one shown in Fig. 6a, ranging from 1.08 to 1.21 times. This showcases that the signatures selected by the greedy algorithm has close performance to the optimal ones.



(a) Cost ratio.

(b) Processing time ratio.

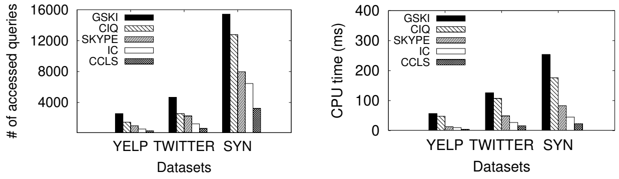
Fig. 6: Signature selection.



(a) Average processing time.

(b) Memory consumption.

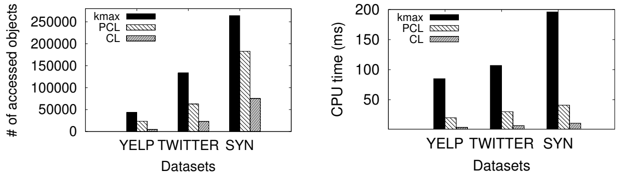
Fig. 7: Overall efficiency.



(a) # of accessed queries.

(b) Average processing time.

Fig. 8: Affected query finder.



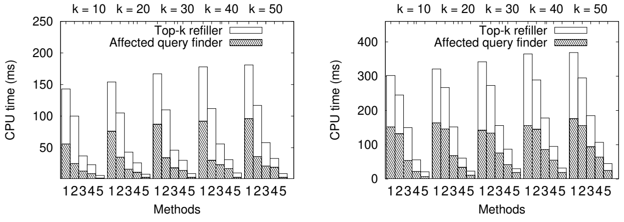
(a) # of accessed objects.

(b) Average processing time.

Fig. 9: Top-k refiller.

D. Query Processing Performance

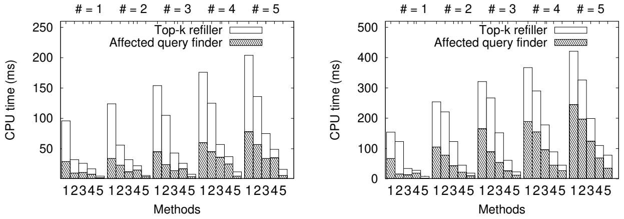
We compare our method with alternative solutions and plot the average query processing times in Fig. 7a. GSKI-kmax is the slowest. CIQ-kmax is faster than GSKI-kmax due to the group pruning. By using prefix filtering, SKYPE-kmax further improves the performance. CCLS-CL outperforms the runner-up, IC-PCL, by 2.4 to 3.3 times speed-up, and reduces the average query processing time to a range of 8.1ms to 34.0ms. We plot memory consumptions in Fig. 7b. GSKI-kmax includes hierarchical grid and inverted index. CIQ-kmax and SKYPE-kmax include quadtree, inverted index, and object buffer. IC-PCL includes grid and cell list. CCLS-CL includes grid, CC links, inverted index built upon l-signatures, and cell list. The memory consumptions of these methods are close, ranging from 1.5 to 5.2 GB. CIQ-kmax is the largest since it keeps all the queries in every leaf node of the quadtree. IC-PCL spends the smallest amount of memory. Next we evaluate AQF and TR separately.



(a) YELP.

(b) TWITTER.

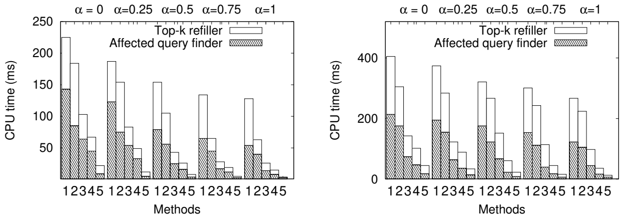
Fig. 10: Varying k.



(a) YELP.

(b) TWITTER.

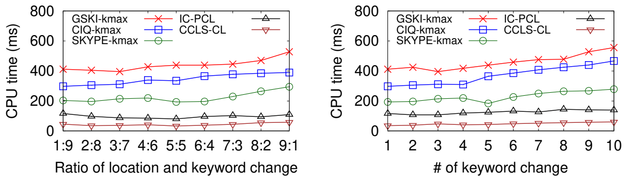
Fig. 11: Varying number of keywords in queries.



(a) YELP.

(b) TWITTER.

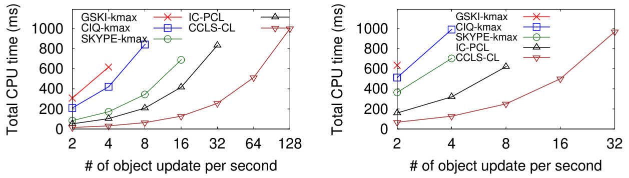
Fig. 12: Varying α.



(a) SYN.

(b) SYN.

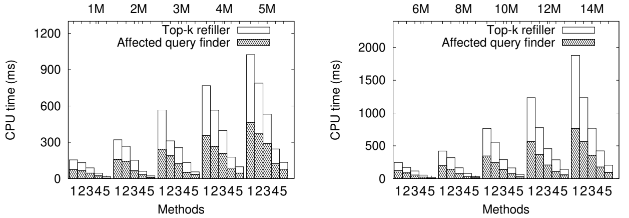
Fig. 13: Effect of object change.



(a) YELP.

(b) TWITTER.

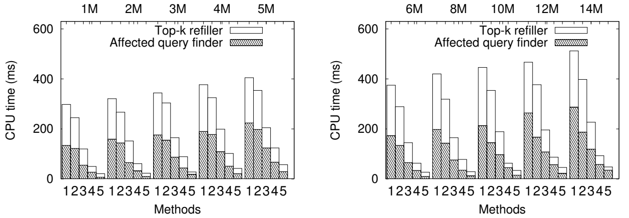
Fig. 14: Varying object update rate.



(a) TWITTER.

(b) SYN.

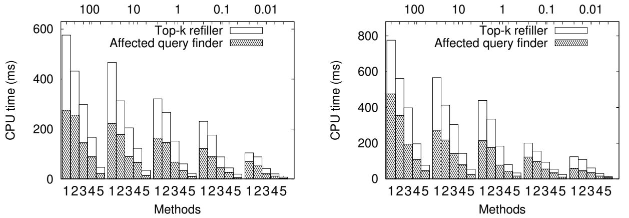
Fig. 15: Varying number of queries $|Q|$.



(a) TWITTER.

(b) SYN.

Fig. 16: Varying number of objects $|O|$.



(a) TWITTER, sum = 7M.

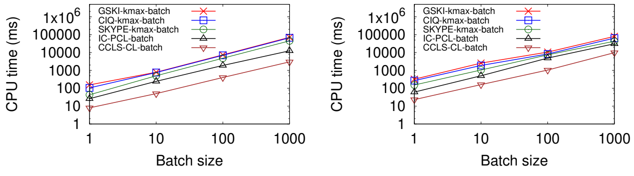
(b) SYN, sum = 20M.

Fig. 17: Varying ratio of $|Q|$ to $|O|$.

For AQF, we first compare the number of queries accessed, results shown in Fig. 8a. GSKI accesses the most number of queries, followed by CIQ, SKYPE, and IC. Thanks to l-signatures, CCLS accesses 50% to 62% queries compared to IC. In Fig. 8b, we plot the running times of AQF. Equipped with CC links and l-signatures, our method is faster than IC by 1.7 to 2.5 times. For TR, we plot the number of objects accessed in Fig. 9a. kmax performs the worst because it maintains a buffer of objects which updates frequently. PCL's object access is moderately smaller. CL performs the best, accessing only 20% to 41% objects compared to PCL. The reduction in object access is converted to less running time in TR, as shown in Fig. 9b. PCL and CL are both drastically faster than kmax. CL reaches a speed-up of 3.7 to 5.0 times over PCL.

E. Scalability

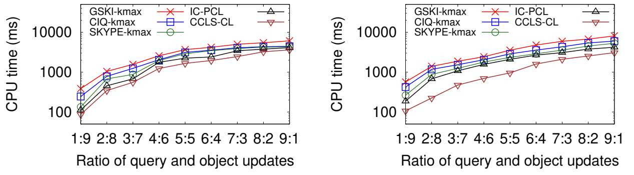
Fig. 10 plots the running times on YELP and TWITTER datasets by varying k from 10 to 50. In the figure, we use 1 – 5 to denote GSKI-kmax, CIQ-kmax, SKYPE-kmax, IC-PCL, and CCLS-CL, respectively. The running times are broken down to AQF and TR. As k increases, all the methods spend slightly more time in AQF and TR. This is because the k-th object score decreases, and thus more cells, queries, and objects are accessed. We vary the number of keywords from 1 to 5 in queries and show the running times in Fig. 11. The trend is similar to varying k, but the effect is more significant. In Fig. 12, we vary α, the weight of spatial similarity, and plot the running times. The running times decrease when we use a larger α. The reason is



(a) YELP.

(b) TWITTER.

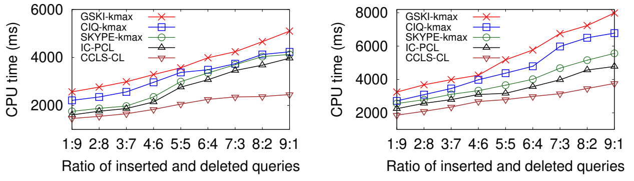
Fig. 18: Batch processing.



(a) YELP.

(b) TWITTER.

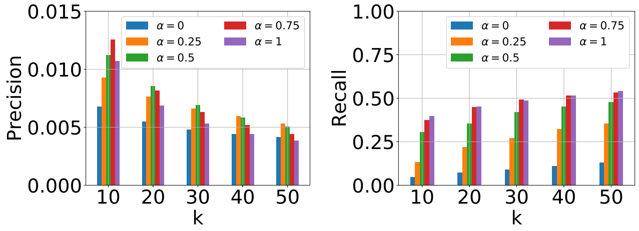
Fig. 19: Varying ratio of query/object updates.



(a) YELP.

(b) TWITTER.

Fig. 20: Varying ratio of inserted/deleted queries.



(a) Precision.

(b) Recall.

Fig. 21: Effectiveness.

that a larger α indicates more weight on spatial information, which has better pruning power than keyword similarity.

We adjust the ratio of location change to keyword change; e.g., 9:1 means location changes 9 times frequently than does keyword. This set of experiments was conducted on the synthetic dataset SYN as it is not supported by the real datsets. The running times are shown in Fig. 13a. The general trend is that GSKI-kmax, CIQ-kmax, and SKYPE-kmax have increasing running time when keyword change is more frequent, while the other two methods slightly fluctuates. The main reason is the first three methods use single keywords to handle keyword similarity, and hence more sensitive to the frequency of keyword change. In contrast, IC-PCL does not index keywords; CCLS-CL is equipped with l-signatures to reduce the time spent on processing keywords, thereby mitigating the influence of keyword change. In Fig. 13b, we show the performance on SYN when varying the number of keywords changed per object update (we use fixed locations here). With more keywords changed in an update, all the methods fluctuate in running time but show a generally increasing trend. This is because more change of keywords leads to more change in the object's score, hence increasing the probability of more query and object access to compute top-k. We vary the update rate (the number of objects updated per second) exponentially and show in Fig. 14 the average running time to process the specified updates per second (2, 4, 8, . . .). We do not plot the running time if it exceeds 1s, since this means that the method cannot keep up with the pace of update. CCLS-CL is able to handle 128 and 32 updates per second on YELP and TWITTER, respectively. The other competitors remarkably trail behind ours: the runner-up, IC-PCL, can only process up to 32 and 8 updates per second on the two datasets.

Figs. 15 and 16 show the running times on TWITTER and SYN datasets when varying the numbers of queries and objects, respectively. All the methods spend more time when we include more queries or objects. The effect of query number is more remarkable because we need to maintain k results and an object buffer (or a cell list) for one additional query, which accounts for more running time than having one more object. In Fig. 17, we fix the sum of query and object numbers (7M on TWITTER and 20M on SYN) and vary their ratio from 100:1 to 1:100. The running times decreases when there are more objects than queries. This is in accord with the results in Figs. 15 and 16, showcasing more effect of query number than object number on query processing performance.

F. Batch Processing

We study the performance on batch processing by monitoring top-k for 100 batch updates, with batch size varying from 1 to 1,000. A batch size of 1 is exactly the case of a single object update. Fig. 18 shows the average processing time per batch on YELP and TWITTER. We compare our method with the other competitors adapted for batch processing. The average processing time increases with the batch size for all the competitors. Thanks to the one-signature-selection-for-all and $2k$-CL techniques tailored to batch processing, CCLS-CL substantially outperforms alternative solutions (note the log-scale). At a batch size of 1,000, all the other competitors spend tens of seconds, while CCLS-CL is faster than the runner-up by 3.1 and 4.2 times, reducing the time to only 2.8s and 9.8s.

G. Query Updates

We consider insertion and deletion of queries and evaluate the performance. We first fix the ratio of insertion to deletion as 1:1, and vary the ratio of query updates to object updates. The running times are plotted in Fig. 19 (note the log-scale). The speed-up of CCLS-CL is smaller than the one without query updates, but still ranges from 1.6 to 2.1 times on YELP and 1.4 to 2.3 times on TWITTER over the runner-up, IC-PCL. The reason for this result is that processing a query update spends more time than processing an object update, mainly due to the initialization of its top-k results. We then fix the ratio of query to object updates to 1:1, and vary the ratio of inserted and deleted queries. The running times are plotted in Fig. 20. Because processing a insertion involves more operations (initialization of top-k and inserting into inverted index), the running time moderately increases when we have more insertions than deletions. CCLS-CL is constantly faster than the alternative methods.

H. *Effectiveness*

We study the effectiveness by a case study of identifying potential customers for e-commerce. We use the Foursqaure check-in dataset [13], which was used in [31]. It contains 227,428 check-ins in New York City, with a total of 1,083 users and 38,333 POIs. We regard each user as an object and each POI as a query. We randomly partition the dataset for k-fold cross-validation ($k = 5$).

- The training set (denoted by $\mathcal{A}$) is a stream of check-ins on which our method processes top-k spatial-keyword queries. We assume that user instantly moves between POIs; i.e., given a check-in (ρ, t) in the dataset, where ρ is the location of a POI, the user instantly moves from its previous location to ρ at t. A POI has an average of 2.09 keywords. For a user's keywords, we retrieve 10 keywords of the recent POIs that the user has checked in (we set the initial state as the first timestamp at which each user has 10 keywords). A POI identifies a user as a potential customer if the user becomes the POI's top-k result.
- We evaluate precision and recall on the testing set (denoted by $\mathcal{B}$). Precision and recall are defined as follows.

$$precision = \sum_{q \in Q} \frac{|R_{q,\mathcal{A}} \cap C_{q,\mathcal{B}}|}{|R_{q,\mathcal{A}}|}.$$

$$recall = \sum_{q \in Q} \frac{|R_{q,\mathcal{A}} \cap C_{q,\mathcal{B}}|}{|C_{q,\mathcal{B}}|}.$$

Q is the set of POIs. $R_{q,\mathcal{A}}$ is the set of users that have become a top-k result of q in $\mathcal{A}$, i.e., the users who are identified as potential customers by the POI q. $C_{q,\mathcal{B}}$ is the set of users that have checked in q in $\mathcal{B}$, i.e., the users who are the customers of the POI q in the ground truth. In other words, precision measures amid the users that have been identified as a POI's potential customer, the percentage of them who have checked in this POI in the ground truth; recall measures amid the users that have checked in a POI in the ground truth, the percentage of them who are identified as the POI's potential customer.

Figure 21 plots the effectiveness measures by varying k and α values. When k moves towards a larger value, the precision decreases and the recall increases. This is expected: a larger k indicates that POIs identify more potential customers and at the same time more false positives. The best choice of α for precision centers around 0.5 and the best α for recall is 1. This means users are equally concerned about spatial proximity and textual relevance if we seek more precise identification, while focusing on textual relevance results in more potential customers identified. The recall is up to 0.54, significantly higher than the precision, which is around 0.01. Note that (1) such precision is common for recommender systems on sparse datasets (227,428 check-ins v.s. 1,083 users × 38,333 POIs), e.g., POI recommendation by temporal matching between POI popularity and user regularity [32], in which the same dataset was used for experiments; (2) we aim at high recall (i.e., identifying more customers), and additional post processing methods can be applied atop of our method to improve precision, e.g., by personalized or time-sensitive recommendation.

XI. Conclusion and Future Work

We proposed an approach to monitoring top-k results for a large set of spatial keyword queries on dynamic objects. Our approach consists of an affected query finder module and a top-k refiller module. We analyzed technical challenges and developed pruning strategies and corresponding data structures to speed up the two modules. Extension to batch processing was discussed. We conducted extensive experiments on real and synthetic datasets. The results showed that our solution significantly outperforms alternative methods in terms of efficiency. Our future work includes improving results for recommender systems and considering more semantics (e.g., by word embeddings).

Acknowledgement

This work was partly supported by JSPS KAKENHI Grant Numbers JP17H06099, JP18H04093, JP19H04114, JP19K12114, and JP19K11979.

References

[1] P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *SIGMOD*, pages 927–938, 2010.

[2] R. Ahuja, N. Armenatzoglou, D. Papadias, and G. J. Fakas. Geo-social keyword search. In *SSTD*, pages 431–450, 2015.

[3] Amazon.com. Amazon prime air. https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011, 2019.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

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

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

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

[8] L. Chen, S. Shang, Z. Zhang, X. Cao, C. S. Jensen, and P. Kalnis. Location-aware top-k term publish/subscribe. In *ICDE*, pages 749–760, 2018.

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

[10] 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.

[11] Y. Dong, H. Chen, and H. Kitagawa. Continuous search on dynamic spatial keyword objects. In *ICDE*, pages 1578–1581, 2019.

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

[13] Foursquare. Foursquare check-in dataset. https://sites.google.com/site/yangdingqi/home/foursquare-dataset, 2019.

[14] Y. Gao, X. Qin, B. Zheng, and G. Chen. Efficient reverse top-k boolean spatial keyword queries on road networks. *IEEE Trans. Knowl. Data Eng.*, 27(5):1205–1218, 2015.

[15] 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.

[16] 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.

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

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

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

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

[21] NYC Open Data. Green taxi trip data. https://data.cityofnewyork.us/Transportation/2016-Green-Taxi-Trip-Data/hvrh-b6nb/, 2016.

[22] Robomart. Self-driving stores. https://robomart.co/, 2019.

[23] 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.

[24] Twitter. Geotagged tweets from the us. https://datorium.gesis.org/xmlui/handle/10.7802/1166, 2016.

[25] P. Wang, C. Xiao, J. Qin, W. Wang, X. Zhang, and Y. Ishikawa. Local similarity search for unstructured text. In *SIGMOD*, pages 1991–2005, 2016.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] D. Yang, D. Zhang, V. W. Zheng, and Z. Yu. Modeling user activity preference by leveraging user spatial temporal characteristics in lbsns. *IEEE Trans. Systems, Man, and Cybernetics: Systems*, 45(1):129–142, 2015.

[32] Z. Yao, Y. Fu, B. Liu, Y. Liu, and H. Xiong. POI recommendation: A temporal matching between POI popularity and user regularity. In *ICDM*, pages 549–558, 2016.

[33] Yelp. Yelp dataset. https://www.yelp.com/dataset, 2019.

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

[35] N. E. Young. Greedy set-cover algorithms. In *Encyclopedia of Algorithms*. 2008.

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

[37] J. Zhao, Y. Gao, G. Chen, C. S. Jensen, R. Chen, and D. Cai. Reverse top-k geo-social keyword queries in road networks. In *ICDE*, pages 387–398, 2017.

[38] 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.