Temporal Geo-Social Personalized Keyword Search Over Streaming Data

ABDULAZIZ ALMASLUKH, King Saud University, Riyadh, Saudi Arabia
YUNFAN KANG and AMR MAGDY, University of California, Riverside

The unprecedented rise of social media platforms, combined with location-aware technologies, has led to continuously producing a significant amount of geo-social data that flows as a user-generated data stream. This data has been exploited in several important use cases in various application domains. This article supports geo-social personalized queries in streaming data environments. We define temporal geo-social queries that provide users with real-time personalized answers based on their social graph. The new queries allow incorporating keyword search to get personalized results that are relevant to certain topics. To efficiently support these queries, we propose an indexing framework that provides lightweight and effective real-time indexing to digest geo-social data in real time. The framework distinguishes highly dynamic data from relatively stable data and uses appropriate data structures and a storage tier for each. Based on this framework, we propose a novel geo-social index and adopt two baseline indexes to support the addressed queries. The query processor then employs different types of pruning to efficiently access the index content and provide a real-time query response. The extensive experimental evaluation based on real datasets has shown the superiority of our proposed techniques to index real-time data and provide low-latency queries compared to existing competitors.

CCS Concepts: • **Information systems** → **Multidimensional range search**; **Stream management**; **Data streaming**; *Query operators;*

Additional Key Words and Phrases: Spatial, temporal, geo-social, real-time, indexing, query processing

ACM Reference format:
Abdulaziz Almaslukh, Yunfan Kang, and Amr Magdy. 2021. Temporal Geo-Social Personalized Keyword Search Over Streaming Data. *ACM Trans. Spatial Algorithms Syst.* 7, 4, Article 20 (August 2021), 28 pages.
https://doi.org/10.1145/3473006

1 INTRODUCTION

The unprecedented popularity of online social media platforms over the past decade combined with the availability of location information through GPS-equipped devices has led to significant

Abdulaziz Almaslukh work has been performed while the author was at the University of California, Riverside.
This work is partially supported by the National Science Foundation, USA, under grants IIS-1849971, SES-1831615, and CNS-2031418.

Authors' addresses: A. Almaslukh, College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia; email: aalmaslukh@ksu.edu.sa; Y. Kang and A. Magdy, Department of Computer Science and Engineering, Center for Geospatial Sciences, University of California, Riverside; emails: ykang040@ucr.edu, amr@cs.ucr.edu.

2374-0353/2021/08-ART20 $15.00
https://doi.org/10.1145/3473006

attention for supporting geo-social queries at scale [6, 7, 16] in order to serve applications efficiently on such big data. These queries are used in various applications and services such as social recommendations [9, 40, 49], community and event detection [11, 24, 46], and urban planning [20]. A major category of these queries is personalized search queries that use the social information to tailor the query answer per the issuing user. For example, a user who is concerned about COVID-19 infections in her social circle wants to find recent posts that contain *coronavirus* or *COVID-19* keywords from her friends in the city of *Los Angeles, California.* To allow finding recent posts at a fine temporal granularity, it is required to manage geo-social data as a data stream. In fact, the modern geo-social data has a streaming nature due to the large number of its data items that arrive every second around the clock. Latest assessments estimate Twitter to receive approximately 8,500 tweets/second [22] while Facebook posts are even an order of magnitude larger in size [17, 22]. This streaming nature has already motivated several streaming queries on this data, such as keyword queries [3, 30, 42], spatial queries [29, 32], and social queries [27, 35], with plenty of applications. Although several geo-social queries, including keyword predicates, have been addressed in the literature [6, 7, 16, 23, 26, 44, 50], querying streaming data combining social, geographical location, and textual information is still an unaddressed challenge.

Geo-social queries have got a little attention in the streaming environments although several applications that are powered by these queries will significantly benefit from the real-time nature of geo-social data, e.g., providing real-time search on friends' posts during emergency situations and detecting real-time events based on friends' updates. In such streaming environments, hundreds of millions of items arrive at high pace every day, which puts major challenges on real-time indexing and query processing based on social, geographical, and textual information. These challenges include sustainable digestion of new data in real-time index structures and exploiting the social information, which is usually complex in structure and huge in size, to serve incoming queries that have certain locations of interest. State-of-the-art techniques [8, 26, 38, 39] are still limited to address these challenges, either for inefficient indexing for real-time data or inefficient query processing navigating highly complex graph structures, which limits using streaming geo-social textual information in scalable applications.

This article introduces scalable real-time indexing and query processing for geo-social personalized search queries over streaming data. The index and query processing design are made to support efficient snapshot queries and can be used as an efficient initial phase for continuous querying modules. We first define two queries that combine three aspects: spatial, temporal, and the social connectivity between users. They are socio-temporal extensions of the two fundamental spatial queries, range query and k-nearest-neighbor query, to effectively serve the streaming data applications that are timely by nature. An example of such queries is to *"find what my friends/friends-of-friends have recently posted in Los Angeles"*, where a spatial range encapsulates *Los Angeles* city boundaries, or *"find what my friends/friends-of-friends post now nearby Tampa, Florida"* in case of hurricane emergency. Such queries are obviously useful for various applications that make use of personalized real-time content, such as improving emergency response by involving the close social circle of individuals or getting personalized recommendations from friends. To limit the query answer to the top relevant items, the queries use ranking functions based on timestamp and discrete social distance, similar in spirit to hop count, to retrieve only the top-k items that satisfy the query predicates. We further extend these queries to incorporate the textual aspect. The extended queries take a set of keywords as input and produce objects that only contain one or more of these keywords. An example of extended keyword search queries could be *"find what my friends/friends-of-friends post now about coronavirus or COVID-19 nearby Los Angeles"* where *"coronavirus or COVID-19"* serves as the keyword set to further filter out the most textually relevant to such a pandemic disease.

In support of these queries in real time, we propose a geo-social indexing framework that distinguishes highly streaming data from relatively stable data. Then, it employs memory-based light indexing for incoming streams and disk-based indexing for stable data. Based on this framework, we propose novel geo-social indexes that effectively organize real-time data for efficient querying based on either pure temporal aspect or combining temporal and textual aspects. The indexes consist of three components: an in-memory spatio-temporal index, an in-disk social index, and an in-memory buffer. During query processing, both in-memory and in-disk data are combined to retrieve relevant data from direct friends in the social graph. If the retrieved data items are less than k, then the query search expands to search indirect friends at one or more levels of social expansions to retrieve the final top-k answer. Due to the awareness of social aspect, the query processor smartly prunes the search space based on social connectivity in addition to spatial, temporal, and textual information. Such multi-dimensional pruning significantly reduces the query response time and reduces contention on the real-time index structure to maintain high real-time data digestion rates.

This work is a significant extension from our previous work [4] to enable keyword search on geo-social streaming data. The extension adds a new query predicate to the original queries definitions. The new predicate takes a set of keywords to produce output that only contains one or more of these keywords. Supporting such new predicate by trivially extending the query processor to employ a keyword filter after getting the results provides unacceptable performance. Such simple filtering leads to processing significantly large number of objects to produce an answer of size k, where k is relatively small. Consequently, keyword support must be inherent in both indexing and query processing modules to enable efficient keyword search. This leads to radical extensions to different modules of this work, both indexing and query processor. Extending these modules is challenging and have different considerations and trade-offs. For example, existing indexes already have three-dimensional structures to efficiently handle spatial, temporal, and social aspects of the data. So, it is not clear if adding the textual within the same structure provides reasonable trade-off between indexing efficiency in real time and fast query processing. In nutshell, this new query predicate introduces several technical challenges to be supported efficiently through existing indexing and query processing. Thus, this extended work addresses these challenges to enable efficient keyword search on geo-social streaming data in real time. Our extended experimental evaluation studies tradeoffs of using existing modules versus the newly proposed extensions.

The extensive experimental evaluation of our proposed techniques on real datasets has shown superiority over competitor techniques that are incorporated from the literature. Using a single machine setting, our indexes can digest up to 220K object/second of streaming data while providing an order of milliseconds query latency for both average and 99% of the queries. In addition, the in-memory component of our proposed indexes consistently maintains low memory usage compared to competitor techniques. Our contributions in this article can be summarized as follows:

- We extend the fundamental spatial queries to define temporal geo-social personalized search queries that retrieve data objects based on spatial, temporal, and social predicates on streaming data in real time.
- We further extend the temporal geo-social personalized search queries to enable keyword search in real time.
- We propose a novel real-time indexing framework that efficiently digests geo-social streaming data based on different attributes.
- We study various considerations and tradeoffs of instantiating the indexing framework for different attribute combinations on real-time indexing.

- We develop query processing techniques that exploit the index content and further prune the search space to provide low query latency.
- We extensively evaluate the proposed techniques compared to existing competitors on real Twitter datasets showing their superiority and effectiveness for streaming environments.

The rest of this article is organized as follows: Section 2 presents the related work. Section 3 presents the problem definition. Sections 4 and 5 detail the proposed geo-social indexing and query processing techniques. Section 6 provides an extensive experimental evaluation. Finally, Section 7 concludes the article.

2 RELATED WORK

There is no current research work that addresses geo-social queries on user-generated streaming data in real time to the best of our knowledge. However, social-aware queries are supported independently on both spatial user-generated data and streaming user-generated data in the literature. This section covers this literature and distinguishes it from our proposed work.

Queries on User-Generated Streaming Data. User-generated streaming data has received significant attention over the past few years due to the popularity of online social media platforms and similar online services. In addition to continuous queries [34, 42] that were the only foci of traditional machine-generated streaming data, user-generated streaming data has been exploited for various applications and snapshot queries, such as geo-textual queries [3, 12–14, 25, 30], location-based search [8, 10, 32], trend detection [1, 18, 36], time-sensitive recommendations [47], and news and topic extraction [19, 37, 43]. In this literature, the spatial and social aspects of the queries are addressed independently. So, geo-textual queries (e.g.,[3], [12], [13], [14], [25], and [30]), and location-based search queries (e.g., [8], [10], and [32]) do not support any social or personalized aspect, and personalized queries (e.g., [26]) do not consider the spatial dimension. A recent attempt to combine both spatial and social dimension is proposed in [38]. However, their solution creates a complete disk-based spatial index for each user, which is extremely expensive for streaming data and cannot even scale to be a baseline approach to compare with. Our work distinguishes itself from existing techniques to be the first to combine both spatial and social aspects in one query while considering streaming environments and a keyword search for both lightweight real-time indexing and efficient query processing. This was not addressed by any of the existing techniques.

Social Queries on Spatial Data. Due to the importance and various applications that benefit from combining social and spatial aspects, several researchers have recently developed indexing and query processing techniques for different geo-social queries (e.g., [2], [6], [7], [16], [35], and [50]). This includes recommending POIs [2, 39, 40, 48, 49], finding cliques [21, 28, 45], finding top-*k* spatial-keyword objects [2, 44], and finding top-*k* influential users [2, 23]. Some of the works (e.g., [2], [35], [44], and [50]) support geo-social keyword search queries. However, none of these techniques address geo-social personalized search queries on streaming data. Thus, our work is distinguished from all existing techniques in multiple ways. First, we are the first to extend geo-social queries and geo-social keyword queries with the temporal aspect due to the nature of streaming data that is the main focus of this article. Second, we are the first to consider lightweight real-time indexing and query processing for geo-social data. This real-time aspect of the streaming environment puts significant overhead on both indexing and query processing, which cannot be handled by any of the existing techniques.

3 PROBLEM DEFINITION

We evaluate the geo-social queries on a streaming dataset D that consists of geo-social objects. Each object $o \in D$ is represented with the four main attributes (uid, loc, $keywords$, $timestamp$), where

Table 1. Content of Objects in Figure 1

UID	OID	Keywords	Timestamp
$u1$	$o1$	Fantastic, Comeback, Play	05-08-2021 20:18:30
$u2$	$o2$	Love, Pineapple, Pizza	05-08-2021 20:18:27
$u3$	$o3$	Sunny, Day, Good, Running	05-08-2021 20:18:23
$u1$	$o4$	Freeway, Traffic, Bad	05-08-2021 20:18:19
$u4$	$o5$	University, Graduation	05-08-2021 20:18:17
$u2$	$o6$	USA, Japan, Summit	05-08-2021 20:18:14
$u5$	$o7$	Airport, Flight, Time, Ready	05-08-2021 20:18:09
$u6$	$o8$	NBA, Lakers, LeBron	05-08-2021 20:18:06

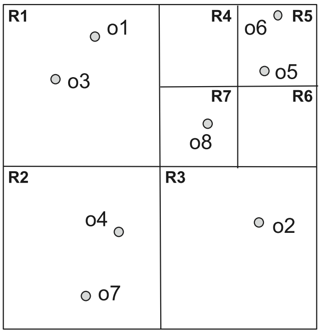
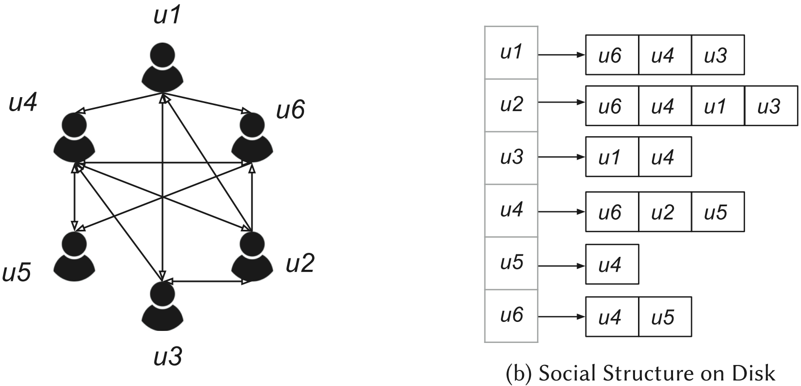


Fig. 1. Spatial Quadtree (SQ).

uid is the identifier of user who posted this object, *loc* is the location where the object is posted in the two-dimensional space represented with latitude/longitude coordinates, *keywords* is the set of keywords extracted from the textual content of the object, and *timestamp* is the time when the user posts the object. D_T is a snapshot of the dataset D at time T, so every object $o \in D_T$ has $o.timestamp \leq T$. Table 1 shows a sample of the dataset that consists of eight objects. Each object, identified by *oid*, is composed of a user id who posted the object, a set of keywords that represent the textual content, a timestamp, and located in the space as shown in Figure 1. In addition, the social connectivity between the users is represented as a hashtable where the <key,value> pair is <user id, list of friend ids>. The social network and the hashtable of the sample are shown in Figure 2. Each entry of the hashtable consists of the given user id as the key, and the list of user's friends ids as the value. We can easily navigate from a user's friends to the friends of friends by expanding the immediate friends and retrieving their friends. This process can be repeated to navigate to higher levels of the social graph. The simplicity of representation and navigation of the social graph helps the query processors to achieve high query throughput, especially in a tight streaming environment.

The two fundamental spatial queries, in particular range query and k-nearest neighbor, that are common in the literature have been extended to support temporal geo-social aspects in this work. The query definitions of the two extended queries are as follows:

Definition 1 (Spatial-Social Temporal Range Query (SSTRQ)). Given q = <user u, spatial range R, integer k, and timestamp T>, and D_T that is a snapshot of the dataset D at time T, SSTRQ retrieves the most recent k objects $o_i \in D_T$, $1 \leq i \leq k$, that are posted within R and are posted by u's friends or friends of friends based on a discrete social distance.



(a) Example Social Network

(b) Social Structure on Disk

Fig. 2. Example of in-disk social structure.

The k objects are ranked based on time to retrieve the most recent objects in D_T from u's direct friends. Because of the overwhelming number of objects, setting k value helps to provide users with the most relevant objects, which makes the answer useful. In addition, limiting answer size to k objects helps to prune the search space. This still serves all applications as it provides the flexibility to adjust the value of k based on the interest of the application to retrieve more results. If q fails to retrieve all k objects from u's friends, the search is expanded to u's friends of friends recursively to retrieve the rest of objects. So, the social relevance of objects in q answer are assessed based on a discrete social distance that takes only integer values (1,2,3, etc.) and no fractional values in between.

This enables scalable query processing on streaming data in real time as detailed in the following sections:

Example 1. Given q1 = <u5,spatial range R, k = 2, T = 05-08-2021 20:18:30>, q1 is an SSTRQ query that finds the two most recent objects (k = 2) from u5's friends or friends of friends that are posted in the area R as shown in Figure 1. According to the hashtable in Figure 2, the only friend u5 has followed is u4; hence, the only object o5 from u4 in Table 1 is included in the result. The objects from the friends of u4, i.e., u2 and u6, are checked and the most recent object, o2, is added to the result. As a result, the answer to the example query is {o5, o2}.

Definition 2 (Spacial-Social Tamporal kNN Query (SSTkQ)). Given q = <user u, spatial point location L, integer k, and timestamp T>, and D_T that is a snapshot of the dataset D at time T, SSTkQ retrieves top-k objects $o_i \in D_T$, $1 \leq i \leq k$, that are posted by u's friends or friends of friends, and ranked based on a spatio-temporal distance F_α from L and T as follows:

$$F_\alpha(o, q) = \alpha \times SpatialScore(o, q) + (1 - \alpha) \times TemporalScore(o, q),$$

where α is a weighting parameter, $0 \leq \alpha \leq 1$, that weighs the relative importance of spatial and temporal scores in the object proximity. $SpatialScore$ and $TemporalScore$ are defined as follows:

$$SpatialScore(o, q) = \frac{distance(o.loc, q.L)}{R_{Max}}$$

$$TemporalScore(o, q) = \frac{q.T - o.timestamp}{T_{Max}}$$

where R_{Max} and T_{Max} are the maximum allowed spatial and temporal ranges for any object, and $distance$ is the spatial distance between the object and the query locations in the Euclidean space.

The social relevance is assessed using the same discrete social distance that is used in SSTRQ for scalability on streaming data in real time.

Example 2. Given q2 = <$u1$,spatial point location $o1.L$, $k = 1$, T = 05-08-2021 20:18:30>, q2 is an SSTkQ query that finds the object o ($k = 1$) ranked by the ranking function $F_\alpha(o, q2)$ from $u1$'s friends or friends of friends. As shown in the hashtable in Figure 2, the friend list of $u1$ is {$u6$, $u4$, $u3$}. According to Table 1, the set of objects posted by the friends of $u1$ is {$o3$, $o5$, $o8$}. Because $o3$ is closer to the specified location and is also newer than the other two objects, $o3$ gets the highest $SpatialScore$ and $TemporalScore$ and hence is ranked the highest by $F_\alpha(o, q2)$. As a result, {$o3$} is returned as the answer to the query.

The two queries are further extended to include keyword predicates. The extended queries are formally defined as follows:

Definition 3 (Spacial-Social Temporal Keyword Range Query (SSTRQ$_{KW}$)). Given q = <user u, spatial range R, integer k, keyword set kw, and timestamp T>, and D_T that is a snapshot of the dataset D at time T, SSTRQ$_{KW}$ retrieves the most recent k objects $o_i \in D_T$, $1 \leq i \leq k$, that are posted within R by u's friends or friends of friends based on a discrete social distance and $o_i.keywords \cap q.kw \neq \phi$.

The k objects out of SSTRQ$_{KW}$ are still ranked based on time to retrieve the most recent objects in D_T from u's direct friends. The social relevance is assessed using the same discrete social distance that is used in SSTRQ. The new addition in SSTRQ$_{KW}$ is the keyword predicate kw. This predicate has a Boolean *OR* conjunction semantic for query keywords. If the keyword set kw of an object o_i contains at least one keyword in the query keyword set $q.kw$, o_i is eligible to be included in the final answer. Keyword similarity is based on exact string matching.

Example 3. Given q3 = <$u5$,spatial range R, $k = 2$, keyword set kw = {"Love", "Watch", "NBA"}, T = 05-08-2021 20:18:30>, q3 is a SSTRQ$_{KW}$ query that retrieves the two most recent objects (k = 2) posted by the friends or the friends of friends of $u5$ containing at least one keyword in the kw set in range R. Because the object $o5$ posted by the only friend of $u5$ from social distance 1 does not contain any of the keywords, the search space is expanded to social level 2, i.e., the friends of friends. Among the objects posted by the friends of $u4$, only the object $o2$ contains the keyword "Love" and the object $o8$ contains the keyword "NBA". As a result, {$o5$, $o2$} is returned as the top-k results, k equals 2.

Definition 4 (Spatial-social Temporal Keyword kNN Query (SSTkQ$_{KW}$)). Given q = <user u, spatial point location L, integer k, keyword set kw, and timestamp T>, and D_T that is a snapshot of the dataset D at time T, SSTkQ$_{KW}$ retrieves top-k objects $o_i \in D_T$, $1 \leq i \leq k$, that are posted by u's friends or friends of friends, ranked based on a spatio-temporal distance F_α from L and T, and $o_i.keywords \cap q.kw \neq \phi$.

Where F_α is the same ranking function that is detailed in the SSTkQ definition, the social relevance is assessed in the same way as well. The new keyword predicate also has a Boolean *OR* conjunction semantic for query keywords, so an object that has any of the keywords is eligible to be included in the final answer.

Example 4. Given q4 = <$u2$,spatial point location $o6.L$, $k = 1$, keyword set kw = {"LeBron", "James", "University"}, T = 05-08-2021 20:18:30>, q4 is a SSTkQ$_{KW}$ query that find the highest ranked object ($k = 1$) by the ranking function $F_\alpha(o, q4)$ from $u2$'s friends or friends of friends. The list of friends for $u2$ is {$u6$, $u4$, $u1$, $u3$}, as shown in Figure 2. According to Table 1, the list of objects from the friends of $u2$ is {$o1$, $o3$, $o4$, $o5$, $o8$}. Among the list of objects from the friends of $u2$, $o5$ contains the keyword "University", and $o8$ contains the keyword "LeBron". Object $o5$ is ranked higher than $o8$ by the ranking function $F_\alpha(o, q4)$ because $o5$ is posted closer to the location specified by the query and is also more recent than $o8$. Because the k is 1, $\{o5\}$ is the result returned.

4 GEO-SOCIAL REAL-TIME INDEXING

This section presents geo-social data indexing in real time. This data is rich with spatial, temporal, textual, and social information. The two main challenges in indexing such rich data in real time are: (1) encoding the incoming information in highly scalable data structures that are efficient for insertions with tens of thousands of data objects each second and (2) removing old data from the main memory to sustain digesting new incoming data objects at all times. Traditional insertion procedures in spatial and social index structures incur significant overhead that limits scalable data digestion. In addition, straight forward deletion procedure that scan every index cell in different spatial regions or different parts of the social graph to expel old data incur significant overhead that will also affect the indexing scalability in real time.

To address these challenges, we introduce a generic indexing framework (Section 4.1) that separates highly dynamic data from relatively-stable data, so real-time data structures are tailored to digest only the needed information in real time to reduce both insertion and deletion overheads. Based on this framework, we propose a scalable index (Section 4.2) that enables efficient handling for geo-social data in real time, and adapt two baseline index structures (Section 4.3) from the literature of spatial and spatial-social indexing. Finally, we extend the proposed index in two different ways to support the keyword queries more efficiently (Section 4.4). The rest of this section details the indexing framework as well as the five indexes.

4.1 Indexing Framework

The proposed indexing framework depends on the observation that incoming geo-social data objects are highly dynamic while the social graph information is relatively static. Each second, tens of thousands of geo-social objects are flowing, which requires real-time digestion. These objects are posted by hundreds of millions of users that are connected to each other with social bonds, represented as a social graph. This social graph is not updated frequently compared to the geo-social objects. In a real Twitter dataset, an active user posts, on average, seven tweets per day [41], which leads to hundreds of millions of tweets every day. However, the number of new friends or unfollowed friends are not even close to this daily number. It is usual not to accept new friends or follow new people for several days, weeks, or even months. Consequently, the frequency of updates in social graph information is way less than the incoming geo-social objects in real time. Our indexing framework exploits this observation to dedicate the necessary resources to index each type of data.

The proposed indexing framework consists of three components: (1) in-memory index that digests streaming geo-social objects in real time, (2) in-disk index that organizes relatively stable social graph information, and (3) in-memory buffer that swaps social graph information from and to the disk index. The in-memory index is equipped with optimized insertion and deletion techniques that minimize the real-time overhead and is able to scale for handling streaming data. As main-memory is a scarce resource, data cannot be digested infinitely with excessive amounts and have to be expelled to a secondary storage on a regular basis. For that reason, the in-memory index employs a temporal duration T_{Max} that indicates the maximum allowed past data to store. T_{Max} is a system parameter and can be adjusted by the administrators based on the available main-memory resources and the streaming rates of incoming data.

The second component is an in-disk index that stores the social graph information. Two reasons are behind storing this information on disk. First, the excessive size of this information consumes significant memory storage that is not frequently utilized, due to the long-tail distribution where the majority of users are inactive in queries [31]. For example, a subset of our experimental Twitter social graph with 3.3 million users consumes approximately 62.5 GB of main-memory as each user has an average of 500 friends. Second, the relative stability of social graph information as discussed earlier in this section. Because of this, the social graph index structure needs infrequent updates, which is not challenging to be handled on the disk storage. However, for query processing, it is inefficient to visit the disk for every retrieval of a user friend list, especially for active users who post frequent queries. This has motivated the third component of our indexing framework, which is the in-memory buffer for social graph information. This component acts like the database buffer, where certain disk pages are swapped in the main-memory buffer from the disk index only when needed. As disk pages keep accumulating in the buffer, it becomes full and needs to evict some of its content to swap in new pages. Eviction policies that are used for the buffer are the same ones that are studied in the literature of database buffer management and operating system virtual memory. We choose to use the famous **least recently used (LRU)** policy in our realization. However, other policies could be used, based on the underlying application requirements.

4.2 SSQ Index

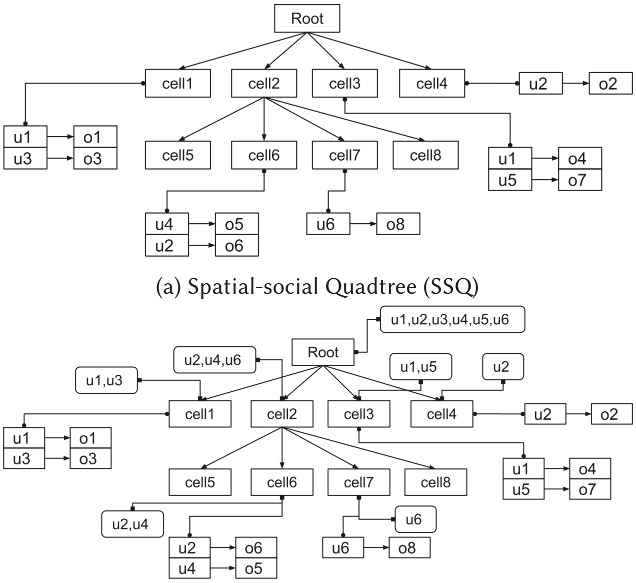
Based on the described framework in Section 4.1, we propose **Spatial-Social Quadtree (SSQ)** index for scalable real-time indexing of geo-social objects without recording the keyword sets. Conformed to the framework, the index has three components: an in-memory component for digesting objects in real-time, a disk-resident component for the social graph indexing, and an in-memory buffer, as described in Section 4.1. This section describes the details of the index structures and update operations for different index components.

Index Structure. The in-memory component adopts a spatial quadtree [5] as a highly-scalable space-partitioning index for real-time data digestion [33]. Spatial quadtree adapts with skewness in spatial distribution and could adapt with dynamic data with low indexing cost in real-time. As a space-partitioning index, it does not need heavy restructuring with changing its data content. In addition, it allows the index cell split and merge operations to be modified to reduce real-time indexing overhead and scale up for high rates of real-time data as shown in [33].

An example of spatial quadtree is depicted in Figure 1 for eight geo-social objects that are presented in Table 1. The tree divides the space into multi-level disjoint cells that either have four or zero children cells. An incoming object is located in the cell that contains its location. A cell is divided into four quadrants only if the number of objects exceeds a specific cell capacity, which is a system parameter that determines the tree height, so a small cell capacity leads to a deeper tree while a large cell capacity generates a shallow tree. Only leaf nodes hold data objects, while intermediate nodes provide routing information. SSQ index extends the quadtree to be aware of the user aspect of the spatial objects. Specifically, each leaf cell is equipped with a hash index structure that organizes the cell's objects based on the issuing users. This hash structure is light for real-time digestion, and still provides effective pruning for the search space, based on the social information. The hash structure uses the user id as a key and the value is a list of objects that are posted by this user order, based on their timestamps. Including the social information within the spatial cell significantly helps the query processor to retrieve candidate objects that could potentially make it to the final answer.

Figure 3(a) depicts an example of the SSQ in-memory index. The depicted index represents the same set of objects that are depicted in Figure 1, and the same quadtree organization, with adding the light hash structure to each leaf node that enables effective social-based pruning while sustains high digestion rates in real time as verified in our experimental evaluation.

The in-disk component of SSQ index stores the social graph represented by a set of adjacency lists. Our social graph representation adopts the famous form that represents users as nodes and



(a) Spatial-social Quadtree (SSQ)

(b) Tightly-coupled Spatial-social Quadtree (TCSSQ)

Fig. 3. Structure of geo-social real-time indexes.

friendship relations as directed edges. The adjacency list representation stores this information as a hash structure that uses the user id as a key and the list of friends as a value for each hash entry. Figure 2 demonstrates an example for a social graph with six users, $u1$ to $u6$. Figure 2(a) shows the high-level graph model for the social relations among the six users while Figure 2(b) shows the adjacency list representation that is stored on the disk-resident index structure. The disk structure consists of two parts, the data part and the index part. The data part stores consecutive blocks of long integer lists that contain the user ids as depicted in Figure 2(b). The index part stores all the distinct user ids, each user id is associated with a disk pointer to the block in which the user friend list is stored. Compared to the data part, the index part is small in size and can be easily loaded during the query processing for efficient access of user information as described in Section 5.

To reduce the overhead of reading back and forth from the disk, the third component of SSQ index is a dedicated in-memory buffer that is utilized to store the retrieved user friend lists from the disk for further recycling during future queries. The in-memory buffer is a hash structure that stores key-value pairs of user ids and friend lists, similar in format to the disk index from which data is retrieved. When the in-memory buffer is full, it adopts the LRU policy to free up content to continue serving incoming queries.

Index Insertion. Insertion in both the in-disk component of SSQ index and its corresponding buffer adopts traditional one-by-one insertion due to the low insertion rates in the stable social structure. On the contrary, the in-memory index component, that adopts a social-aware quadtree, incurs an excessive insertion rate as tens of thousands of objects arrive every second. The traditional insertion procedure, which navigates the tree hierarchy for each incoming object and inserts it in the corresponding cell, does not scale enough to cope with such a high insertion rate. To overcome this problem, we employ a batch insertion process that collects a few seconds worth of data in a temporary buffer and inserts them as one batch in the quadtree structure. During the buffering, a **minimum bounding rectangle (MBR)** is maintained around the location of incoming objects. Then, the MBR boundaries are compared to the index cell boundaries, instead of comparing location of each object, and the tree navigation is performed based on this cheap comparison. With thousands of objects buffered, thousands of comparison operations are saved, which significantly boost the digestion performance and allows the index to ingest streaming data with high arrival rates.

As the tolerable buffering delay depends on the underlying application, the buffering time is adjustable by the system administrators to meet the application needs. The main motivating use cases for our techniques work on streaming user-generated data, such as social media and other content that is generated by human users online. In this context, a few seconds of delay is usually tolerable. For example, when users search major social media platforms, the most recent results are usually posted a few seconds ago. It is worth noting that the high rate of streaming data in these applications enables a very small buffering delay while still buffering thousands of data items. So, a typical buffering delay of 1-2 seconds is enough to enable scalable indexing in real time, which is a reasonable delay that fits most of the mainstream applications.

The index insertion and the queries can be handled concurrently while still maintaining high real-time data ingestion through employing a single-writer-multiple-readers concurrency model as detailed in [32] and [3]. This model is slightly modified in this work to enable queries to expire data that is beyond T_{Max} time units as pointed out in index deletion below. The data that has the potential for concurrent access from reader threads is already expired and removed from the index shortly after, so they minimally affect the real-time index update operations.

The speculative cell-splitting module [33] is used to reduce insertion and query-processing time. A leaf cell is split if it exceeds its capacity and the objects in the leaf cell will span at least two quadrants.

Index Deletion. To sustain digesting incoming data in the scarce memory resources, the in-memory index expels objects that are older than T_{Max} time units ago to the disk, where T_{Max} is a system parameter that is based on the availability of memory resources and arrival rates of the underlying streaming data. To expel this data, a straightforward way is to exhaustively iterate over all index cells, either every few time units or when a certain memory budget fills up, and clean up all expired data objects that are older than T_{Max}. However, such exhaustive and frequent cleaning process puts an overhead on real-time operations of the index. To avoid such overhead, we employ a combination of regular and periodic cleaning processes that are lighter than the exhaustive cleaning and still sustain memory consumption. The regular cleaning is piggybacked on the real-time insertion and querying, so whenever an index cell is accessed for either insertion or query processing, the accessed entries are checked for expired content to be expelled from main-memory. This reduces the cleaning overhead as it shares the index traversal overhead with the other operations.

This regular cleaning process does not guarantee to expel all the expired data proactively as it depends on the spatial distributions of both data and queries, so some index cells might be left without cleaning due to infrequent access to those cells. To address this, we employ a light periodic cleaning that goes over all index cells every T_{Max} time units. For each cell, if it is not cleaned during the past T_{Max} time units, which means no insertions happened during this period, all the cell content is wiped as all objects are expired. Otherwise, the cell is skipped. This process is very light and mainly addresses cells that are infrequently accessed. In addition, it can be easily invoked in a separate thread to reduce the contention over index cells in real time. We adopt the lazy cell merging strategy to manage the leaf nodes after deletion. If a leaf node becomes empty after the deletion, the siblings of the leaf node are examined. If two of the siblings are empty, the content of the third sibling is moved to their parent node and the four leaf nodes are removed. The lazy cell merging saves 90% of the split empirically and merge operations and reduces the index update overhead significantly. The details for the lazy cell merging is given in [33] and it is not considered a novel contribution for this article.

4.3 Baseline Indexes

In addition to our proposed SSQ index (Section 4.2), we adopt two baseline indexes based on the proposed indexing framework that is described in Section 4.1. The two baseline indexes are alternatives to address the supported queries based on existing techniques in the literature. The two baseline indexes are **Spatial Quadtree (SQ)** and **Tightly-Coupled Spatial-Social Quadtree (TCSSQ)**. The rest of this section describes each index and highlights its differences compared to the proposed SSQ index.

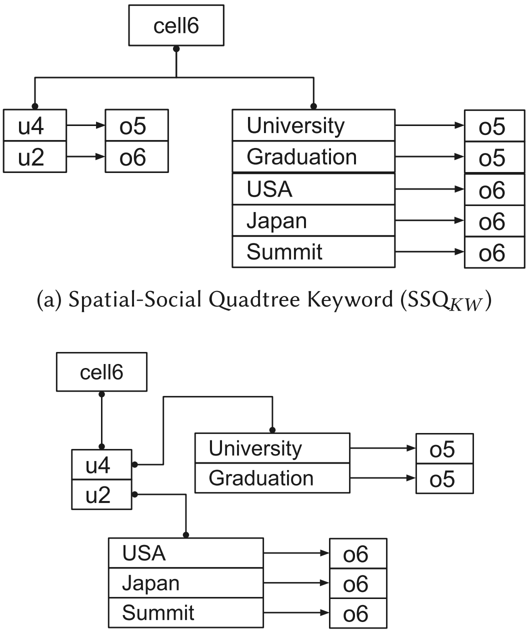
(1) *Spatial Quadtree (SQ).* This index has a similar structure to the SSQ index with the exception of the in-memory index component that adopts a pure spatial quadtree structure without any extended structures to organize the data based on the posting users. Figure 1 shows an example of the spatial quadtree index. It is worth noting that all data objects in the leaf nodes are sorted based on their arrival timestamp at no additional cost due to the nature of the streaming data that comes ordered by time. For the index insertion and deletion, the same procedures that are developed for SSQ index are used in SQ index with the exception of navigation the leaf nodes content that does not have the hash structure anymore. So, inserted data are appended to a long list of chronologically ordered objects, and all the cleaning processes are performed on the same list, which reduces the real-time indexing overhead while increases the query processing overhead as will be detailed in Sections 5 and 6.

(2) *Tightly Coupled Spatial-Social Quadtree (TCSSQ).* This index has a similar structure to the SSQ index with the exception of the in-memory index component that includes extra user information in all intermediate and leaf nodes of the quadtree structure instead of having a hash structure in only leaf nodes. Specifically, each leaf node C has an additional list of users $C.L_u$ who posted in the spatial region of C. Then, the content of $C.L_u$ is replicated to the parent nodes up to the root node. So, the root's L_u has all the users who posted in any region, and each intermediate node has a list of all users who posted in the sub-tree that is rooted in this intermediate node. This organization is a modified version of [44] that is suitable for real-time indexing. This is built based on the core ideas of the IR-tree structure [15]. Figure 3(b) depicts an example of TCSSQ index for the eight objects of Table 1. Each node, including root, intermediate, and leaf nodes, has an additional list $C.L_u$ of users who posted in the node C spatial region.

The additional user lists L_u affect the index insertions and deletions in real time. On insertion, after the insertion procedure is performed in node C as described for SSQ index, the posting user id uid is added to $C.L_u$. To this end, uid is searched in $C.L_u$ using binary search. If uid does not exist in $C.L_u$, it is inserted into the ordered list; otherwise, $C.L_u$ remains intact. Then, the same process repeats for parent nodes' L_u until it propagates to the root node. On index deletion, object deletions are performed for certain user entries in the node's hash structure. For each user entry, if the list of objects remains non-empty, i.e., there are still remaining objects for this user in the node, $C.L_u$ remains intact. On the contrary, if the list of objects becomes empty, i.e., the deleted objects are the last objects for this user in the node, then the user id is removed from $C.L_u$. Then, the removal checks are propagated to parent levels of the tree. For C's parent L_u, the three siblings nodes of C are checked. If uid exists in any of their L_u lists, then the parent's L_u remains intact. If uid does not exist in any of these lists, then uid is removed from the parent's L_u, and the removal check is propagated to the higher levels, up to the root node.

4.4 Keyword Indexing

This section presents the geo-social indexes that incorporate the keywords while indexing the geo-social objects in order to process keyword-extended geo-social queries in streaming data environment efficiently. First, we present **Spatial-Social Quadtree Keyword (SSQ$_{KW}$)**, and then **Spatial-Social Quadtree 4D (SSQ$_{4D}$)**.



(a) Spatial-Social Quadtree Keyword (SSQ$_{KW}$)

(b) Spatial-Social Quadtree 4D (SSQ$_{4D}$)

Fig. 4. Geo-social keyword index structure for cell 6 based on Figure 3(a).

(1) Spatial-Social Quadtree Keyword (SSQ$_{KW}$). This index is adopted from Spatial-Social Quadtree (*SSQ*) since the experiments have shown its superior performance compared with baseline indexes. It has a different index structure than the in-memory component while it has exactly the same components of *SSQ*, including the in-disk and in-memory buffer components. More specifically, it attaches a hash index called the *inverted keyword index* for each leaf cell of *SSQ* in-memory quadtree in order to prune the objects based on the keywords effectively. This makes each leaf cell have two separate hash indexes; one for the social as in *SSQ* where the <key,value> pair is <user id, list of friend ids>, and the additional one for the keywords where the <key,value> pair is <keyword, list of objects>. The objects are organized based on the social information as in *SSQ* and additionally based on the keywords. Therefore, the index structure helps the query processor to significantly reduce the query latency with minimal overhead on the digestion rate and the memory resource. Figure 4(a) shows the index structure of SSQ$_{KW}$ for a cell with two hash indexes. The leaf cell has both the social information that represented with a hash index that organizes the objects based on the issuing users and another hash index that indexes the objects based on the keywords appeared in the objects. For insertion and deletion, the same procedures of *SSQ* detailed in Section 4.2 are applied to maintain a high digestion rate for the new incoming data. However, additional operations are needed to insert/remove the objects to/from the accompanying keyword inverted index to be consistent with the user hash index. Any object inserted/removed to/from the user hash index must be inserted/removed accordingly to/from the keywords hash index. So, the object is being inserted into the posting user's list of the user index and inserted also into all keyword lists, which contain the object's keywords, of the keyword index. Once the object is removed from the user index, as being older than T_{Max} time units ago as detailed in Section 4.2, all the lists of objects where the object's keywords are the keys shall be retrieved in order to remove the object in order to be synchronized with the user index.

(2) *Spatial-Social Quadtree 4D (SSQ_{AD}).* SSQ_{4D} is another index structure adopted from Spatial-Social Quadtree (*SSQ*) that adds the keyword dimension differently to support the keyword queries more efficiently. Only the in-memory index structure is different from *SSQ* while the other components remain the same. At the leaf cell of *SSQ* tree, SSQ_{4D} indexes the objects based on the social information first, then for each indexed user u, u points to the inverted keyword list that organizes the objects based on the keywords appeared in the objects. The hash index has the structure <key,value> where the key is the user id, and the value is another nested hash index where the <key,value> pair is <keyword, list of objects>. In other words, each user u has her dedicated inverted keyword index which is only indexing the objects that been posted by u in the given cell. Thus, the query processor takes the advantage of the index structure to prune the objects spatially, socially, and textually at the same time. Figure 4(b) depicts the index structure of SSQ_{4D} for a cell. The cell has the user index as the first level, and, for each user in the user index, has her own inverted keywords index as the second level. For insertion and deletion, SSQ_{4D} follows similar steps as *SSQ* detailed in Section 4.2, taking into account the objects are being indexed in a nested hash index based on the keywords. Thus, the object is being inserted into multiple entries of the u keyword hash index based on the object keywords. When removing the object which is older than T_{Max}, all entries of the user keyword hash index should be accessed to remove the object from these lists.

5 QUERY PROCESSING

This section details the query processing of the four queries that are defined in Section 3 exploiting the proposed SSQ index, the baseline SQ and TCSSQ indexes, and the keyword-extended indexes SSQ_{KW} and SSQ_{4D} that are introduced in Section 4. In Section 5.1, we introduce a high-level query processing framework that is generic for all indexes. Sections 5.2 and 5.3 detail the query processing of SSTRQ and SSTkQ queries, respectively, without involving the textual features. Then, Sections 5.4 and 5.5 explain the processing of SSTRQ_{KW} and SSTkQ_{KW}, respectively, in SQ, SSQ, SSQ_{KW}, and SSQ_{4D}.

5.1 Query Processing Framework

Our query processor consists of two generic steps:

(1) *Step 1*: Given the user id *uid* of the query issuing user u, step 1 retrieves a list of friends $u.L_f$ that contains a set of user ids for u's direct friends. To this end, the in-memory buffer of the social graph is checked with the key value *uid*. If it exists, $u.L_f$ is directly retrieved from the buffer. Otherwise, the in-disk social index is accessed in a traditional way to retrieve $u.L_f$ to the in-memory buffer. If the in-memory buffer is full, the least recently used (LRU) replacement policy is used to free up some of the buffer content. Then, $u.L_f$ is fed to step 2 of the query processor.

(2) *Step 2*: Given a list of friends L_f, that is retrieved in step 1, and spatio-temporal predicates, in step 2, the query processor accesses the in-memory spatial index to retrieve the top-k objects based on the query semantic and the underlying index structure. The specifics of this step is different for each <query, index> combination, as detailed in the rest of this section.

If the execution of these two steps retrieves k objects, then they are considered a final query answer and returned to the user. If the computed answer has less than k, the search is expanded recursively beyond u's social level 1 (direct friends) to social level 2 (friends of friends) or higher social levels until k objects are retrieved. To this end, the two steps are repeated for each user id in L_f for expansion to social level 2, and the same repeats for higher social levels.

5.2 SSTRQ Query Processing

This section details the specifics of step 2 of Section 5.1 for SSTRQ query. In this step, the query processor retrieves the most recent k objects within a spatial region R, per the query definition, that are posted by users in the friend list L_f that is computed in step 1. The rest of this section details this procedure using SSQ, SQ, and TCSSQ indexes.

SSTRQ in SSQ Index. SSTRQ query is processed on three phases in SSQ index: (a) spatial retrieval, (b) social filtering, and (c) temporal pruning. First, the spatial retrieval phase navigates the quadtree to retrieve the tree nodes that intersect with the query region R. Second, for each node, the social filtering phase accesses the hash index and retrieve lists of objects that are associated with user ids in the friend list L_f. Each of these lists is ordered based on the timestamp due to the streaming nature of incoming objects. Third, the retrieved lists are enqueued in a priority queue Q that orders lists based on their most recent object. Then, the lists are traversed in Q order to compute an initial answers Ans of k objects. Based on Ans, a temporal boundary T_k is computed as the timestamp of the kth object in Ans. Any object older than T_k cannot be part of the final answer. So, T_k is used as a temporal pruning boundary to process the rest of the objects in Q. In specific, each list in Q is retrieved in order. Then, the list's objects are traversed in time order. If the current object $o.timestamp < T_k$, then o is added to Ans replacing the kth object, and T_k is updated. Otherwise, o is skipped. Once we reach an object $o.timestamp \geq T_k$, the rest of the list is pruned as no more objects can make it to the final answer. This repeats for all lists in Q before Ans is returned as a final query answer.

SSTRO in SQ Index. In SQ index, SSTRQ is processed using the first and third phases, spatial retrieval and temporal pruning, that are used in SSQ index. As SQ index does not include any user information, the social filtering phase cannot be employed. So, the list of objects in each quadtree node is scanned to select the objects associated with user ids in the friend list L_f and fed directly to the temporal pruning phase that produces the final answer using the same procedure that is described above.

SSTRQ in TCSSQ Index. In TCSSQ index, SSTRQ is processed using the same three phases that are used in SSQ index, with an extended social filtering phase. In particular, TCSSQ index maintains extra user list information $C.L_u$ in each quadtree node C. So, the social filtering phase goes through two stages. The first stage is intersecting the user friend list $u.L_f$ with the node user list $C.L_u$. If the intersection is empty, then C and all its descendants are immediately pruned. Otherwise, C is considered for the second stage that is exactly similar to the social filtering phase in SSQ index. The other two phases, spatial retrieval, and temporal pruning, remains identical to the ones in SSQ index.

5.3 SSTkQ Query Processing

This section details step 2 of the query processing framework that is presented in Section 5.1 for SSTkQ query. This query retrieves the closest k objects, based on a spatio-temporal distance function F_α, nearby a point location L and relative to a query timestamp T that are posted by users in the friend list L_f that is computed in step 1, in accordance with the query definition in Section 3. The rest of this section details the query processing using SSQ, SQ, and TCSSQ indexes.

SSTkQ in SSQ Index. SSTkQ query is processed on two phases in SSQ index: (a) computing initial answer and (b) answer refinement. The first phase navigates the quadtree structure to the tree node C that contains the query location L. Then, initial k objects that are associated with users in the friend list L_f are retrieved as an initial answer Ans. If C has less than k objects posted by L_f users, then neighbor nodes are checked until Ans has k objects.

The second phase uses the kth F_α score of the initial answer (namely, $F_{\alpha,k}$) as a refinement boundary to compute the final answer Ans so any object with $F_\alpha \geq F_{\alpha,k}$ cannot make it to the final answer. This could be done in a traditional way by visiting all nodes within the maximum spatial range R_{Max} and check objects that are associated with L_f. However, with excessive amounts of data, this could be very expensive and has high query latency. To compute the final answer efficiently, a spatio-temporal pruning procedure is employed to significantly reduce the number of checked objects. To this end, two pruning boundaries are calculated and updated throughout the second phase based on the equation of F_α: a spatial boundary R_u and a temporal boundary T_u. The spatial upper bound R_u is calculated by assuming zero temporal score in the spatio-temporal ranking function, so $R_u = \frac{F_{\alpha,k}}{\alpha} \times R_{max}$. Similarly, the temporal upper bound T_u is calculated by assuming zero spatial score in spatio-temporal ranking function, so $T_u = q.time - \frac{F_{\alpha,k}}{1-\alpha} \times T_{max}$. Any object or cell that are outside R_u and T_u can be safely pruned. So, neighboring quadtree nodes to location L are visited in spatial order with R_u, and objects of each node are checked as long as they are within T_u. With each new object added to *Ans*, $F_{\alpha,k}$ is updated and then R_u and T_u are updated accordingly. So, the pruning boundaries are continuously tightened, which reduces the total number of checked objects and significantly reduces the query latency. When all nodes and objects within R_u and T_u are exhausted, *Ans* is returned as a final answer.

SSTkQ in SQ Index. In SQ index, SSTkQ is processed using the same two phases as in SSQ index with exception to user filtering in quadtree nodes. As SQ index does not include any user information, the list of objects in each quadtree node is used as a whole and fully scanned for filtering objects that are posted by L_f users.

SSTkQ in TCSSQ Index. In TCSSQ index, SSTkQ is processed using the same two phases that are used in SSQ index, with an extended user filtering step. As the TCSSQ index maintains extra user list information $C.L_u$ in each quadtree node C, when a quadtree node is accessed, the user friend list $u.L_f$ is intersected with the node user list $C.L_u$. If the intersection is empty, then C and all its descendants are immediately pruned. Otherwise, C is considered for further processing as described in the two phases of SSQ index.

5.4 SSTRQ$_{KW}$ Query Processing

This section explains the query processing of SSTRQ$_{KW}$ which includes the keywords as predicates for the four indexes, *SQ*, *SSQ*, SSQ$_{KW}$, and SSQ$_{4D}$. The query processor retrieves the most recent k objects within a spatial region R, and the objects contain the keywords query. The candidate objects are posted by friends of the query issuer.

SSTRQ$_{KW}$ in SQ and SSQ Indexes. Since *SQ* and *SSQ* indexes do not support the keyword pruning as the indexes are not aware of the presence of the keywords, we adopted the simple on-the-fly keyword filtering that examines the candidate objects for the presence of the given query keywords. Specifically, objects that satisfy the spatial and social predicates are retrieved as detailed in Section 5.2. Then, before the object is added to the answer list, the query processor checks for the keywords' presence by applying the on-the-fly keyword filter. If any query keyword overlaps with the object text, the query processor will add the object to the answer list to consider it for further processing; otherwise, the object will not be selected.

SSTRQ$_{KW}$ in SSQ$_{KW}$. The query processor generally follows the same phases as SSTRQ in *SSQ*, which are detailed in Section 5.2 with some modifications. First, the query processor retrieves the objects that contain the query keywords from the keyword inverted index. If there is no object in the keyword index, the query processor stops processing the cell. Second, the query processor performs the social filtering to retrieve the objects that are posted by the query issuer's friends and exist in the list of objects that was retrieved from the keyword index from the previous step with the same steps as *SSQ* query processor mentioned in Section 5.2. Thus, the objects will be added to the initial answer *Ans*. The query processor will refine the initial answer *Ans* with the same logic as in *SSQ*. This will expedite the process of retrieving the candidate objects that contain the query keywords by exploiting the additional hash index for the keyword indexing.

SSTRQ$_{KW}$ in SSQ$_{4D}$. In SSQ$_{4D}$, the query processor performs the same *SSQ* phases as detailed in Section 5.2 with an additional phase called *keyword filtering*. Instead of retrieving all objects from the given user, the query processor accesses the keyword inverted index that each user has and retrieves objects that contain only the keywords query. Spatial and temporal pruning are employed to prune objects that would not make it to the final answer in the same way detailed before.

5.5 SSTkQ$_{KW}$ Query Processing

This section explains the query processing of SSTkQ$_{KW}$ which includes the keywords as predicates for the four indexes. This query retrieves the closest k objects that contain the keywords query, based on a spatio-temporal distance function explained in Section 3.

SSTkQ$_{KW}$ in SQ and SSQ Indexes. The query processor is similar to the query processor of *SQ* and *SSQ* explained in Section 5.3. However, on-the-fly keyword filtering is employed to retrieve objects that contain the keywords query similar to the way described in Section 5.4. Therefore, any objects that did not pass the keyword filtering will not be considered for the initial answer list *Ans*.

SSTkQ$_{KW}$ in SSQ$_{KW}$. The query processor retrieves the list of objects that contain the keywords query by accessing the keyword hash index. Then, the query processor utilizes the underlying index structure to retrieve the objects, from the user hash index, that socially overlap with the query issuer's friends list and intersect with the list of objects which was obtained from the previous step. The other steps and the pruning techniques are similar to the query processor of *SSQ* that is explained in Section 5.3.

SSTkQ$_{KW}$ in SSQ$_{4D}$. In SSQ$_{4D}$, the query processor performs similar steps as those detailed in Section 5.3. However, the query processor does not retrieve all the objects. It retrieves only objects that contain the keywords query by making the use of the keyword inverted index that is associated with every user entry. The pruning techniques are similar to the *SSQ* query processor.

6 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of geo-social real-time indexing and query processing that are discussed in previous sections. Section 6.1 explains the experimental settings. Sections 6.2–6.4 evaluate indexing scalability, memory consumption, and query evaluation, respectively, for SSTRQ and SSTkQ queries. Section 6.5 gives the evaluation for the keyword search queries SSTRQ$_{KW}$ and SSTkQ$_{KW}$.

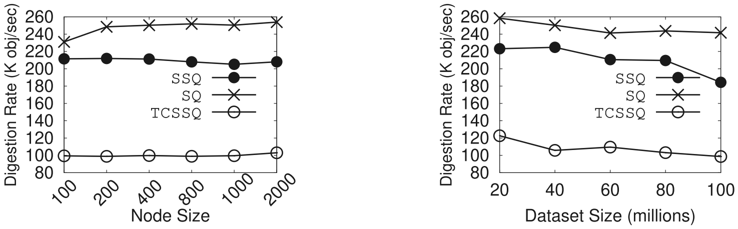
6.1 Experimental Setup

We evaluate the indexes that are discussed in Section 4 for indexing scalability, storage overhead, and query processing. The proposed *Spatial-Social Quadtree* index is denoted as *SSQ*, its keyword extensions denoted as *SSQ$_{KW}$* and *SSQ$_{4d}$*, the baseline *Spatial Quadtree* index is denoted as *SQ*, and the *Tightly Coupled Spatial-Social Quadtree* index is denoted as *TCSSQ*, a modified version of [44] for real-time operations. Our parameters include quadtree node size, dataset size, query answer size k, query range, the space-time weighting parameter α, and the maximum allowed temporal range T_{Max}. Unless mentioned otherwise, the default node size is 2000, dataset size is 80 million objects, k is 100, query range is 50 km, α is 0.2, R_{Max} is 500 km, T_{Max} is one day, number of keywords is 2, and buffer size is 500K entries. The two keywords are selected randomly from the keyword set of the dataset for each keyword query. Our performance measures include index digestion rate (the average number of indexed objects per second), index memory footprint, and query latency. All experiments are based on Java 8 implementation and using an Intel Xeon(R) server with CPU E5-2637 v4 (3.50 GHz) and 128-GB RAM running Ubuntu 16.04.

Evaluation Datasets and Query Workloads. We have collected 6+ billion geotagged tweets from public Twitter Streaming APIs over the course of 5 years. Then, five datasets, of sizes 20, 40, 60, 80, and 100 million tweets, are composed for our evaluation. Each tweet is represented with a

Table 2. Evaluation Dataset Statistics

Dataset	20M	40M	60M	80M	100M
Users	3,379,403	4,589,750	5,323,808	5,862,339	6,319,263
Avg. Friends	531	513	504	497	492



(a) Varying Node Sizes

(b) Varying Dataset Sizes

Fig. 5. Indexing scalability.

latitude/longitude coordinates that represent either an exact location or a centroid of a place, e.g., a city or a landmark. Users of all tweets have been extracted from each of the five datasets. The data includes only the number of friends of each user and not the actual friend list. Thus, we randomly generate a list of friends for each user, where the majority are close to her location while the rest are scattered around the world. Table 2 summarizes the number of users and the average number of friends in each dataset. In order to generate the query workload, we randomly select a thousand users, and their home locations are the query points. For keyword queries, a hundred users are randomly selected and two keywords are randomly selected for a given user from her nearby home location. A random word from the tweet textual content is associated as a keyword.

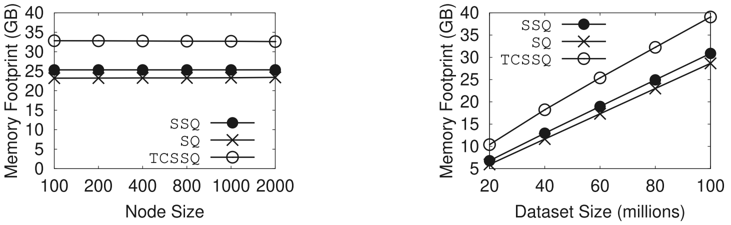
6.2 Indexing Scalability

This section evaluates the scalability of the real-time indexing measured as the number of objects being digested in a second. Figure 5(a) shows the indexing scalability with different quadtree node size. *SQ* can digest on average 250K objects/sec which is the highest among the three indexes. *SSQ* digestion rate is reduced to 210K objects/sec, due to incorporating social information in the index structure, which still maintains 84% of *SQ* digestion rate and digests an order of magnitude higher than the Twitter rate. On the other hand, *TCSSQ* has the lowest digestion rate of 100K objects/sec due to the overhead of summarizing all sub-tree social information. It is though noticeable that different node sizes have no real impact on the digestion rate.

Figure 5(b) shows the impact of different dataset sizes on the digestion rate. The digestion rate slightly decreases when the number of objects increases for all indexes due to the larger index contents, which makes it heavier to digest new data. However, the overall reduction is still acceptable. For example, *SSQ* digests 220K objects/sec with 20 million objects and 190K objects/sec with 100 million objects, which represents 14% reduction of digestion rate and both are still an order of magnitude higher than the Twitter rate.

6.3 Memory Consumption

Figure 6 shows the memory consumption for the three indexes with varying the quadtree node size (Figure 6(a)) and varying dataset size (Figure 6(b)). Varying node size in Figure 6(a) does not significantly affect the memory consumption for all the three indexes despite an order of magnitude higher node capacity, which leads to a significantly lower number of index nodes. This



(a) Varying Node Sizes

(b) Varying Dataset Sizes

Fig. 6. Memory footprint.

shows the minor effect of the index nodes' memory on storage overhead as the majority of memory consumed for data that is being stored inside the nodes. *SQ* consumes the lowest memory, 22 GB, while *SSQ* consumes a slightly higher memory resource, 24 GB, since the index structure keeps more information about the social aspect. *TCSSQ* consumes the highest memory resource, 33 GB, with different index node sizes. The additional social information of *TCSSQ* index structure increases the memory overhead by ~50% of the baseline *SQ* index.

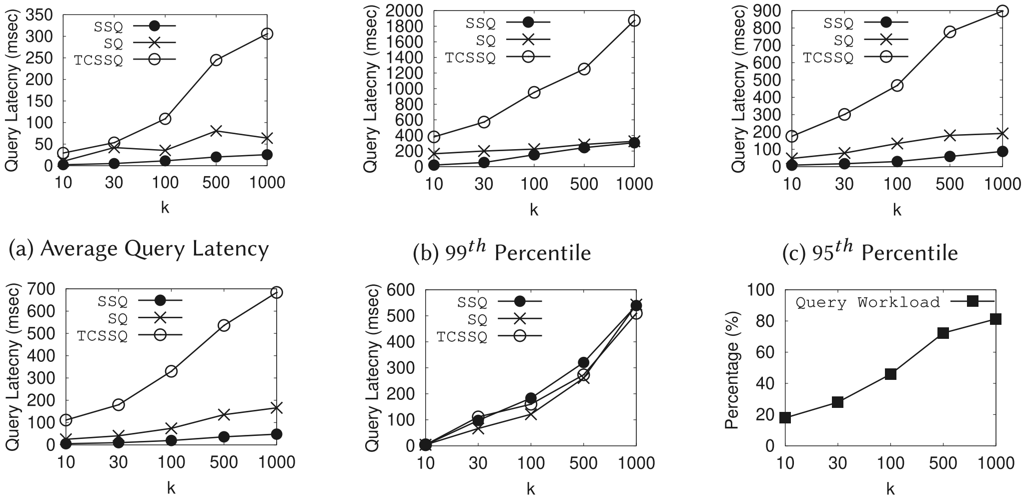
Varying the dataset size in Figure 6(b) affects the memory resources to be increased linearly for all alternatives. For example, *SSQ* consumes 7 GB when the dataset size is 20 million objects, and, when the dataset size triples, *SSQ* consumes 19 GB. The same pattern repeats for *SQ* and *TCSSQ*, where *TCSSQ* always consumes the largest memory. This also confirms that the majority of the memory resources are being consumed by the data that resides in the main-memory.

6.4 Query Evaluation

This section evaluates the query processing of the **Spatial-Social Temporal Range Query (SSTRQ)** and the **Spatial-Social Temporal kNN Query (SSTkQ),** named for the short range query and the kNN query, respectively. The query latency is presented as an average and percentiles, e.g., the 99% percentile latency that shows the maximum query latency for 99% of the queries.

(a) SSTRQ Query Evaluation:

Effect of Varying k. Figure 7 shows the effect of varying k on range query latency, both in-memory and disk processing. Figure 7(a) shows in-memory range query latency measured in milliseconds (msec) for all alternatives. Generally, query latency increases with increasing k due to the additional processing needed for getting a larger answer. However, the latency of *TCSSQ* is significantly higher than the other two alternatives. After monitoring the statistics, we find that the average number of tree nodes visited per query is 175 in our query workload. *TCSSQ* checks whether the friends list $C.L_f$ of each visited node C intersects with the friends list $u.L_f$ of the given user u and prunes 55 nodes on average. Although the search space is reduced by more than 1/4 through social pruning, the social pruning leads to great overhead and causes the *TCSSQ* to have much higher latency than the other two alternatives. As a result, even though this process is effective in disk-based processing of traditional queries, in streaming environments, this process increases the real-time overhead tremendously. As shown in the figure, our proposed *SSQ* index performs the best with 2-msec latency at k = 10, and increases to 25 msec at k = 1000. The *SSQ* index combines both social-aware pruning and lightweight structure that is suitable for real-time environments. The *SQ* index has no social awareness, so it is three times slower than the *SSQ* index on average. It starts with 10-msec latency at k = 10, and it increases steadily to reach 65 msec at k = 1000. The superiority of the *SSQ* index is further confirmed by measuring the 99th, 95th, and 90th



(a) Average Query Latency

(b) 99^{th} Percentile

(c) 95^{th} Percentile

(d) 90^{th} Percentile

(e) Disk Overhead

(f) Social Expansion

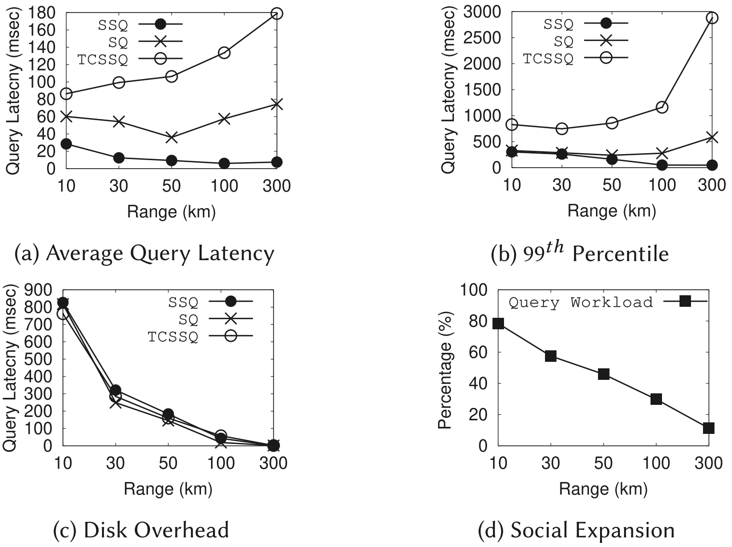
Fig. 7. Range query latency with varying k.

percentile latency as depicted in Figures 7(b), 7(c), and 7(d), respectively. *SSQ* constantly performs the best in terms of query latency, and the advantage is even obvious in Figures 7(c) and 7(d).

Figure 7(e) shows the disk overhead to retrieve the users' friends or friends of friends in order to retrieve the k objects for the given user. All indexes need to access the disk to fetch the social data. Therefore, all alternatives perform similarly, with increasing latency with larger k value, as all indexes use the same disk-based social structure. The increase with k value is explained by the percentage of the query being expanded beyond the first social level (direct friends) as shown in Figure 7(f). The larger k, the less probability that direct friends can satisfy the query answer, and hence expansion to higher social levels is necessary.

Effect of Varying Query Range. Figure 8(a) shows the average query latency with varying query range from 10 km to 300 km. *SSQ* index still performs the best among the other alternatives. Both *SQ* and *TCSSQ* indexes have an increasing latency with the increasing range due to the larger search space. On the contrary, the query latency of *SSQ* drops with increasing range. As *SSQ* employs both temporal and social pruning, the more cells, the more recent initial answer, which in turn produces a tight temporal upper bound. The temporal pruning uses this tight bound to terminate processing very early in many cells. In addition, the social pruning enables us to process only the posting lists that are socially connected to the query issuer, which prunes a significant number of objects that do not contribute to the answer. At the 10-km range, *SSQ* processes queries with an average of 27-msec latency, while at the range of 300 km, this latency drops four times to 7 msec. On the other hand, *SQ* has an almost stable performance with varying ranges as it only employs the temporal pruning, while *TCSSQ* performs the worst although it employs both the temporal and social pruning for the same reasons that were discussed before. Figure 8(b)shows the 99th percentile latency, which confirms the superiority of *SSQ* over all alternatives.

Figures 8(c) and 8(d) show the correlation between disk overhead and the percentage of queries being expanded to higher social levels. Clearly, the disk overhead decreases when the expansion percentage decreases. With small spatial ranges, the probability to retrieve k objects from direct friends is small, and hence the majority of queries expand. This significantly decreases with increasing range.



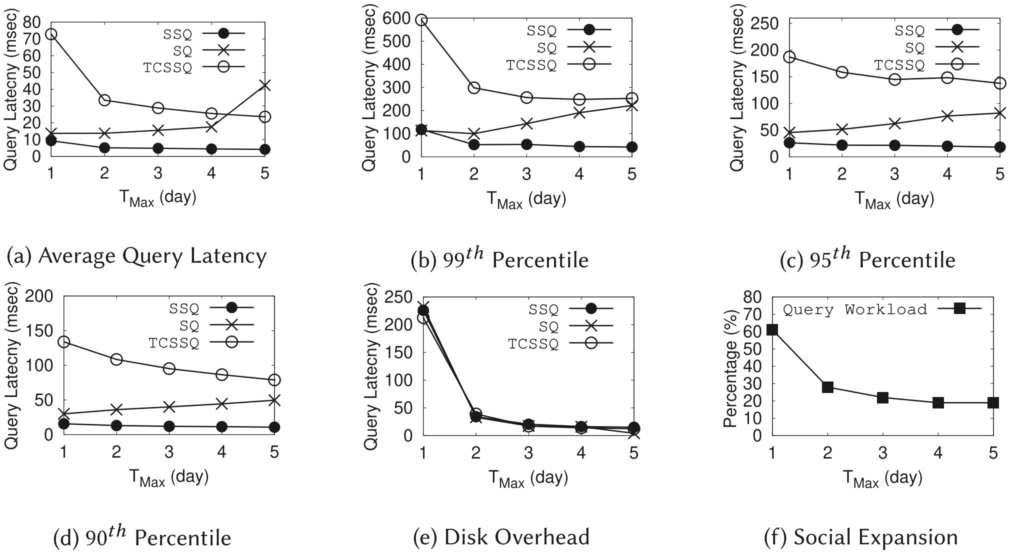
(a) Average Query Latency

(b) 99^{th} Percentile

(c) Disk Overhead

(d) Social Expansion

Fig. 8. Range query latency with varying range query.



(a) Average Query Latency

(b) 99^{th} Percentile

(c) 95^{th} Percentile

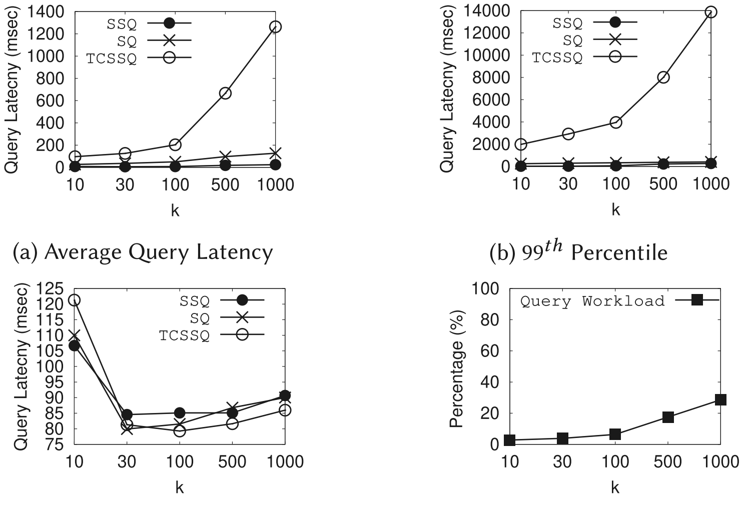
(d) 90^{th} Percentile

(e) Disk Overhead

(f) Social Expansion

Fig. 9. Range query latency with varying T_{Max}.

Effect of Varying T$_{Max}$. Figure 9 illustrates the effect of varying T_{Max} on range query latency, both in-memory and disk processing. Figure 9(f) shows the percentage of queries being expanded to higher social levels with varying T_{Max} from 1 day to 5 days. Obviously, the expansion percentage decreases with the increase of the T_{Max}. When T_{Max} increases, more data objects become available in the main memory and the number of objects associated with each user increases on average. As a result, it is easier to retrieve all the k results from the friends with social distance 1 without expanding to higher social levels. Due to the reduction of the expansion to higher social levels when T_{Max} increases, the disk overhead is also reduced, as shown in Figure 9(e). The correlation



(a) Average Query Latency

(b) 99^{th} Percentile

(c) Disk Overhead

(d) Social Expansion

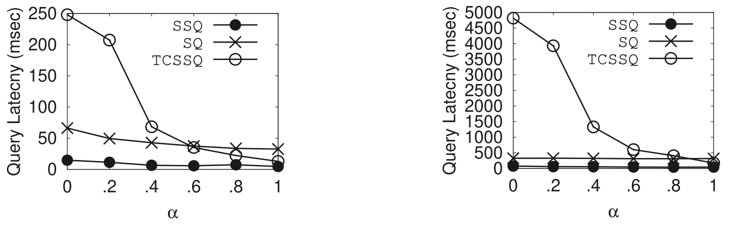
Fig. 10. kNN latency with varying k.

between disk overhead and the social expansion percentage is similar to the previous discussions. Figure 9(a) shows the average query latency with varying T_{Max}. *SSQ* still performs the best among the alternatives and *TCSSQ* is the worst when T_{Max} is 1 day to 4 days for the same reasons that were discussed before. Both *SSQ* and *TCSSQ* benefit from the reduction of the social expansion rate as T_{Max} increases. The overhead for loading the social information to perform the search on higher social levels is largely reduced and both alternatives have a decreasing average query latency when T_{Max} increases. However, the average query latency increases for *SQ*. When more objects become available as T_{Max} increases, the search space for *SQ* is increased because *SQ* cannot perform the social pruning as the *SSQ* and *TCSSQ* do. The increase in the cost by the refinement procedure for *SQ* is more significant than the benefit introduced by the reduced social expansion rate. As a result, the average query latency increases for *SQ* and it performs worse than *TCSSQ* when T_{Max} is set to 5 days. The 99th, 95th, and 90th percentile query latency for the three alternatives in Figures 9(b), 9(c), and 9(d) show the same trend as the average query latency.

(b) SSTkQ Query Evaluation:

Effect of Varying k. Figure 10(a) shows the in-memory query latency with varying k. *SSQ* index performs consistently better than the other alternatives due to its three-dimensional pruning on temporal, spatial, and social dimensions. At $k = 10$, *SSQ* has an average query latency of 9 msec, which increases with larger k to 25 msec at $k = 1000$. This is fifty times better than *TCSSQ* due to its social pruning overhead that is not suitable for real-time processing. On the contrary, *SQ* is slower three times compared to *SSQ* due to lack of social pruning. Such behavior remains the same for the 99th percentile of queries, as shown in Figure 10(b), which shows the superiority of *SSQ* in all cases. For the disk overhead, all alternatives incur almost the same latency as shown in Figure 10(c) due to using the same disk structure. Also, the percentage of socially expanded kNN queries, depicted in Figure 10(d), are much less than range queries since range queries are restricted by a spatial range, which obligates to expand the search to higher social levels often.

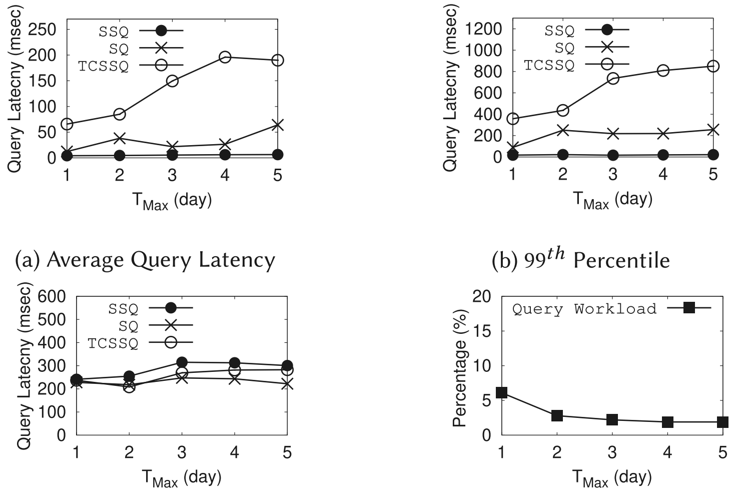
Effect of Varying α. Figure 11(a) shows the effect of varying α that controls the relative importance of the spatial and temporal scores in the spatio-temporal distance. As the figure shows, the



(a) Average Query Latency

(b) 99^{th} Percentile

Fig. 11. kNN latency with varying α.



(a) Average Query Latency

(b) 99^{th} Percentile

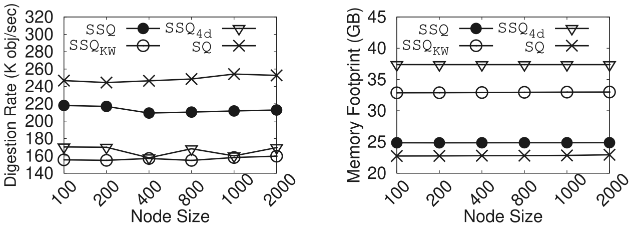
(c) Disk Overhead

(d) Social Expansion

Fig. 12. kNN latency with varying T_{Max}.

α value has a great impact on the query performance, especially for *TCSSQ* index. When only the temporal score is important (at α = 0), all indexes hit their highest query latency because the query processor has to cover a larger search region. With increasing α, the query latency gradually drops to the lowest point for all the indexes when only the spatial score is important (at α = 1). For all values of α, *SSQ* performs the best, while *TCSSQ* performs the worst up to $\alpha < 0.6$. Then, *TCSSQ* performs better than *SQ* after $\alpha \geq 0.6$. The key reason behind this behavior is the number of cells that need to be processed is huge with small α, and *TCSSQ* is very sensitive to the number of cells as it checks for overlap with long user lists. This number decreases as the query region shrinks due to the importance shifts to the spatial closeness. Figure 11(b) confirms similar behavior and *SSQ* superiority on the 99th percentile of queries. For different values of α, the disk overhead is almost stable (approximately 80 msec) for all alternatives except with α = 0 where very few queries expand the search space, which makes the disk overhead very minimal with a few milliseconds.

Effect of Varying T_{Max}. Figure 12(a) shows the average in-memory query latency with varying T_{Max}. As the figure shows, *SSQ* performs better than the other alternatives while *TCSSQ* performs the worst. Query latency tend to increase for all three alternatives. As T_{Max} increases, the density of the object increase both spatially and temporally. Although all three alternatives adopt the spatial



(a) Digestion Rate (b) Memory Footprint

Fig. 13. Indexing overhead.

and temporal pruning technique and the search space is reduced to a large extent, more objects are checked as the density of the objects increases. As a result, the average in-memory query latency increases when T_{Max} increases. The 99th percentile query latency shown in Figure 12(b) confirms a similar trend. Figure 12(c) shows that the disk overhead is stable for all alternatives but the latency caused by the disk overhead is not very low. This is because there are 0.8% to 0.9% failed queries for all values of T_{Max}. In these rare cases, the social network stored on the disk is loaded for multiple times until the queries fail after exploiting the search space. However, because there are only a few failed queries, the percent of queries that are expanded to higher social levels is low for all T_{Max} values in accordance with Figure 12(d).

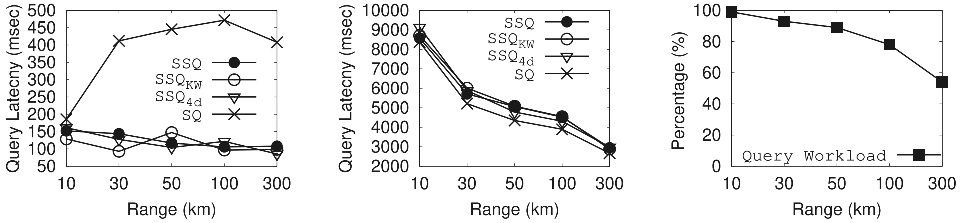
6.5 Keyword Search Evaluation

This section presents evaluation of geosocial keyword search on real-time indexing (*SQ*, *SSQ*, SSQ$_{KW}$, and SSQ$_{4d}$) and query processing of the keyword-extended queries SSTRQ$_{KW}$ and SSTkQ$_{KW}$ as discussed in previous sections. The evaluation focuses on the impact of the keywords query on the digestion rate, memory consumption, and the query latency.

Digestion Rate and Memory Consumption. We evaluate the digestion rate and the memory consumption for varying node sizes for geo-social keyword indexes. Figure 13(a) shows the digestion rate for the four indexes. Clearly, *SQ* and *SSQ* digest more objects than SSQ$_{KW}$ and SSQ$_{4d}$, since the former indexes do not take into account the overhead of indexing keywords. The overhead of indexing keywords is still acceptable for real-time application as both indexes can digest more than 160K and 170K objects/sec on average for SSQ$_{KW}$ and SSQ$_{4d}$, respectively. Figure 13(b) shows the memory consumption of the indexes. SSQ$_{KW}$ and SSQ$_{4d}$ consume the highest memory resources with 37 GB and 33 GB, respectively. The node size does not significantly affect the memory consumption as the data dominates the memory resources rather than the underlying indexing structure.

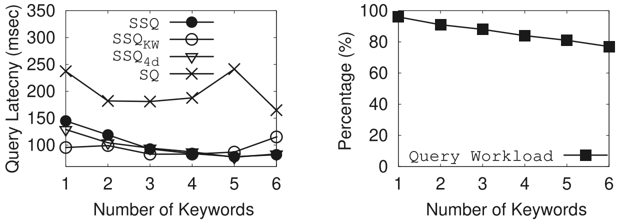
Query Evaluation. We evaluate query processing of both extended queries using different indexes.

(a) *SSTRQ$_{KW}$ Query Evaluation.* Figure 14 shows the the performance of the four indexes for range query with keywords varying the spatial ranges. Figure 14(a) shows that *SSQ$_{KW}$* performs slightly better than the other alternatives while *SQ* performs the worst, since the *SQ* index is not aware of neither the social aspect nor the keyword dimension. This becomes obvious when the spatial range increases when query latency of *SQ* increases significantly while the other steadily decreases. Both *SSQ* and *SSQ$_{4d}$* perform about the same with varying spatial ranges in spite of the fact that the latter is equipped with the keywords indexing. Nevertheless, the social filtering and the temporal pruning are the dominant factors for pruning. Figures 14(b) and 14(c) can draw the same conclusion as Section 6.4 for range query processing without keywords. Figure 15 shows the



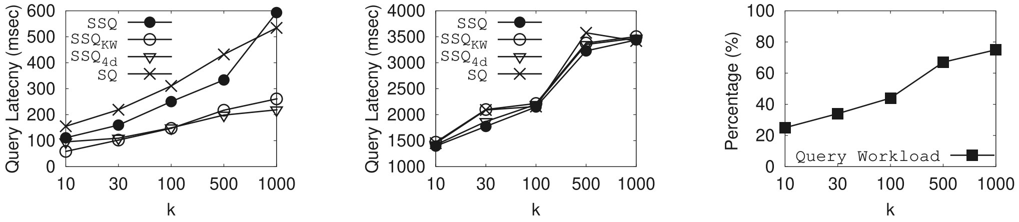
(a) Average Query Latency (b) Disk Overhead (c) Social Expansion

Fig. 14. Range query latency with varying range query.



(a) Average Query Latency (b) Social Expansion

Fig. 15. Range query latency with varying numbers of keywords.

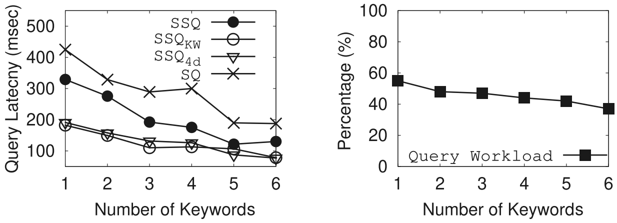


(a) Average query latency (b) Disk Overhead (c) Social Expansion

Fig. 16. kNN latency with varying k.

impact of the number of keywords on the range query. Both Figure 15(a)and Figure 15(b) show a general trend when the number of keywords is increasing, the query latency is decreasing along with the social expansion. Therefore, the query processor can find the candidate objects quickly with the increasing number of keywords.

(b) *SSTkQKW Query Evaluation.* Figure 16 depicts the *k*NN query latency performance for the geo-social keyword indexes. Clearly, the indexes that are equipped with inverted keyword indexes preform significantly better than the indexes that are not aware of the keyword dimension as shown in Figure 16(a). The difference becomes even more obvious when the *k* value increases. More specifically, SSQ_{KW} and SSQ_{4d} perform two times better on average than *SQ* and *SSQ*. Thus, the keywords indexing effect is very obvious in the query processing as it gives an edge over the indexes that do not support the keywords indexing. For the disk overhead and social expansion, the same conclusions can be drawn as explained in Section 6.4 and as shown in Figures 16(b) and 16(c). Figure 17 shows the impact of the number of keywords on the *k*NN query. Both Figures 17(a) and 17(b) show a similar pattern as the range query with increasing number of keywords.



(a) Average query latency (b) Social Expansion

Fig. 17. kNN latency with varying numbers of keywords.

7 CONCLUSION

This article defined temporal geo-social queries on streaming data as extensions for the fundamental spatial k-nearest neighbor (kNN) and range queries. It further extended these queries to support the keyword search feature. To address these queries, we proposed a generic indexing framework for real-time geo-social data that digests and indexes highly dynamic data in main-memory and organizes stable social information in a disk-based structure. Based on this framework, we proposed spatial-social quadtree (SSQ) index and two keyword-aware variants that are lightweight to handle real-time data efficiently, while providing scalable query response for both kNN and range queries. In addition, we adopted two baseline index structures based on the proposed indexing framework. The experimental evaluation on real datasets has clearly shown the superiority of our proposed indexes for both real-time indexing and query processing. For keyword search, SSQ index and its keyword-aware variants provide better performance on streaming data compared to the baseline SQ index. Meahwhile, SSQ maintains a light indexing by using the essential indexing components in a novel way to handle streaming data. We see the novelty in the design of the SSQ index and its variants and consider this as the main contribution of this paper. SSQ performs worse than its keyword extensions for query latency while performs better for indexing overhead. However, the querying loss in SSQ for keyword predicates still makes it reasonable for supporting keyword predicates without extra indexing overhead. On the other hand, the indexing overhead of its keyword variants is still reasonable to support high-velocity streaming data. This shows the impact of the high-level indexing framework that effectively distinguishes dynamic data from stable data and enables various instantiations to perform efficiently in streaming environments.

REFERENCES

[1] Hamed Abdelhaq, Christian Sengstock, and Michael Gertz. 2013. Eventweet: Online localized event detection from Twitter. In *Proceedings of VLDB* (2013).

[2] Ritesh Ahuja, Nikos Armenatzoglou, Dimitris Papadias, and George J. Fakas. 2015. Geo-social keyword search. In *Proceedings of SSTD*.

[3] Abdulaziz Almaslukh and Amr Magdy. 2018. Evaluating spatial-keyword queries on streaming data. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*.

[4] Abdulaziz Almaslukh and Amr Magdy. 2019. Temporal geo-social personalized search over streaming data. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 189–198.

[5] Walid G. Aref and Hanan Samet. 1990. Efficient processing of window queries in the pyramid data structure. In *Proceedings of the SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.

[6] Nikos Armenatzoglou, Ritesh Ahuja, and Dimitris Papadias. 2015. Geo-social ranking: Functions and query processing. *VLDB Journal* (2015).

[7] Nikos Armenatzoglou, Stavros Papadopoulos, and Dimitris Papadias. 2013. A general framework for geo-social query processing. In *Proceedings of VLDB* (2013).

[8] Jie Bao, Mohamed F. Mokbel, and Chi-Yin Chow. 2012. Geofeed: A location-aware news feed system. In *Proceedings of ICDE*.

[9] Jie Bao, Yu Zheng, and Mohamed F. Mokbel. 2012. Location-based and preference-aware recommendation using sparse geo-social networking data. In *Proceedings of GIS.*

[10] Ceren Budak, Theodore Georgiou, Divyakant Agrawal, and Amr El Abbadi. 2013. Geoscope: Online detection of geo-correlated information trends in social networks. In *Proceedings of VLDB* (2013).

[11] Junghoon Chae, Dennis Thom, Harald Bosch, Yun Jang, Ross Maciejewski, David S. Ebert, and Thomas Ertl. 2012. Spatiotemporal social media analytics for abnormal event detection and examination using seasonal-trend decomposition. In *Proceedings of IEEE VAST.*

[12] Lisi Chen, Gao Cong, and Xin Cao. 2013. An efficient query indexing mechanism for filtering geo-textual data. In *Proceedings of SIGMOD.*

[13] Lisi Chen, Gao Cong, Xin Cao, and Kian-Lee Tan. 2015. Temporal spatial-keyword top-k publish/subscribe. In *Proceedings of ICDE.*

[14] Lisi Chen, Yan Cui, Gao Cong, and Xin Cao. 2014. SOPS: A system for efficient processing of spatial-keyword publish/subscribe. *PVLDB* 7, 13 (2014).

[15] Gao Cong, Christian S. Jensen, and Dingming Wu. 2009. Efficient retrieval of the top-k most relevant spatial web objects. In *Proceedings of VLDB* (2009).

[16] Tobias Emrich, Maximilian Franzke, Nikos Mamoulis, Matthias Renz, and Andreas Züfle. 2014. Geo-social skyline queries. In *Proceedings of DASFAA.*

[17] FB Stats 2019. The Top 20 Valuable Facebook Statistics. https://zephoria.com/top-15-valuable-facebook-statistics/. May 2019.

[18] Wei Feng, Chao Zhang, Wei Zhang, Jiawei Han, Jianyong Wang, Charu Aggarwal, and Jianbin Huang. 2015. STREAMCUBE: Hierarchical spatio-temporal hashtag clustering for event exploration over the Twitter stream. In *Proceedings of ICDE.*

[19] Liangjie Hong, Amr Ahmed, Siva Gurumurthy, Alexander J. Smola, and Kostas Tsioutsiouliklis. 2012. Discovering geographical topics in the twitter stream. In *WWW.*

[20] Desislava Hristova, Matthew J. Williams, Mirco Musolesi, Pietro Panzarasa, and Cecilia Mascolo. 2016. Measuring urban social diversity using interconnected geo-social networks. In *Proceedings of WWW.*

[21] Qian Huang and Yu Liu. 2009. On geo-social network services. In *Proceedings of the 2009 17th International Conference on Geoinformatics.*

[22] Internet Live Stats 2019. Internet Live Stats 2019. http://internetlivestats.com/. May 2019.

[23] Jinling Jiang, Hua Lu, Bin Yang, and Bin Cui. 2015. Finding top-k local users in geo-tagged social media data. In *Proceedings of ICDE.*

[24] Ryong Lee and Kazutoshi Sumiya. 2010. Measuring geographical regularities of crowd behaviors for twitter-based geo-social event detection. In *Proceedings of SIGSPATIAL LSBN Workshop.*

[25] Guoliang Li, Yang Wang, Ting Wang, and Jianhua Feng. 2013. Location-aware publish/subscribe. In *Proceedings of KDD.*

[26] Yuchen Li, Zhifeng Bao, Guoliang Li, and Kian-Lee Tan. 2015. Real time personalized search on social networks. In *Proceedings of ICDE.*

[27] Yafei Li, Rui Chen, Jianliang Xu, Qiao Huang, Haibo Hu, and Byron Choi. 2015. Geo-social k-cover group queries for collaborative spatial computing. *In Proceedings of TKDE* (2015).

[28] Weimo Liu, Weiwei Sun, Chunan Chen, Yan Huang, Yinan Jing, and Kunjie Chen. 2012. Circle of friend query in geo-social networks. In *Proceedings of DASFAA.*

[29] Wei Liu, Yu Zheng, Sanjay Chawla, Jing Yuan, and Xie Xing. 2011. Discovering spatio-temporal causal interactions in traffic data streams. In *Proceedings of SIGKDD.*

[30] Amr Magdy, Louai Alarabi, Saif Al-Harthi, Mashaal Musleh, Thanaa M. Ghanem, Sohaib Ghani, and Mohamed F. Mokbel. 2014. Taghreed: A system for querying, analyzing, and visualizing geotagged microblogs. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.*

[31] Amr Magdy, Rami Alghamdi, and Mohamed F. Mokbel. 2016. On main-memory flushing in microblogs data management systems. In *Proceedings of ICDE.*

[32] Amr Magdy, Mohamed F. Mokbel, Sameh Elnikety, Suman Nath, and Yuxiong He. 2014. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. In *Proceedings of ICDE.*

[33] Amr Magdy, Mohamed F. Mokbel, Sameh Elnikety, Suman Nath, and Yuxiong He. 2015. Venus: Scalable real-time spatial queries on microblogs with adaptive load shedding. *IEEE Transactions on Knowledge and Data Engineering* 28, 2 (2015), 356–370.

[34] Ahmed R. Mahmood, Ahmed M. Aly, and Walid G. Aref. 2018. FAST: Frequency-aware indexing for spatio-textual data streams. In *Proceedings of ICDE.*

[35] Shunya Nishio, Daichi Amagata, and Takahiro Hara. 2017. Geo-social keyword top-k data monitoring over sliding window. In *Proceedings of DEXA.*

[36] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. 2010. Earthquake shakes twitter users: Real-time event detection by social sensors. In *Proceedings of WWW.*

[37] Jagan Sankaranarayanan, Hanan Samet, Benjamin E. Teitler, Michael D. Lieberman, and Jon Sperling. 2009. Twitter-Stand: News in tweets. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.*

[38] Ammar Sohail, Muhammad Aamir Cheema, and David Taniar. 2018. Social-aware spatial top-*k* and skyline queries. *Comput. J.* (2018).

[39] Ammar Sohail, Ghulam Murtaza, and David Taniar. 2016. Retrieving top-*k* famous places in location-based social networks. In *Proceedings of the Australasian Database Conference.*

[40] Panagiotis Symeonidis, Alexis Papadimitriou, Yannis Manolopoulos, Pinar Senkul, and Ismail Toroslu. 2011. Geo-social recommendations based on incremental tensor reduction and local path traversal. In *Proceedings of the SIGSPATIAL International Workshop on Location-Based Social Networks.*

[41] Twitter by the Numbers 2019. Twitter by the Numbers: Stats, Demographics & Fun Facts. https://www.omnicoreagency.com/twitter-statistics/.

[42] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Wei Wang. 2015. Ap-tree: Efficiently support continuous spatial-keyword queries over stream. In *Proceedings of the ICDE.*

[43] Hong Wei, Jagan Sankaranarayanan, and Hanan Samet. 2018. Enhancing local live tweet stream to detect news. In *Proceedings of the SIGSPATIAL LENS Workshop.*

[44] Dingming Wu, Yafei Li, Byron Choi, and Jianliang Xu. 2014. Social-aware top-*k* spatial keyword search. In *Proceedings of MDM.*

[45] De-Nian Yang, Chih-Ya Shen, Wang-Chien Lee, and Ming-Syan Chen. 2012. On socio-spatial group query for location-based social networks. In *Proceedings of SIGKDD.*

[46] Hongzhi Yin, Zhiting Hu, Xiaofang Zhou, Hao Wang, Kai Zheng, Quoc Viet Hung Nguyen, and Shazia Sadiq. 2016. Discovering interpretable geo-social communities for user behavior prediction. In *Proceedings of ICDE.*

[47] Quan Yuan, Gao Cong, Zongyang Ma, Aixin Sun, and Nadia Magnenat Thalmann. 2013. Time-aware point-of-interest recommendation. In *Proceedings of SIGIR.*

[48] Jia-Dong Zhang and Chi-Yin Chow. 2013. iGSLR: Personalized geo-social location recommendation: A kernel density estimation approach. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems..*

[49] Jia-Dong Zhang and Chi-Yin Chow. 2015. GeoSoCa: Exploiting geographical, social and categorical correlations for point-of-interest recommendations. In *Proceedings of SIGIR.*

[50] Jingwen Zhao, Yunjun Gao, Gang Chen, Christian S. Jensen, Rui Chen, and Deng Cai. 2017. Reverse top-*k* geo-social keyword queries in road networks. In *Proceedings of ICDE.*

Received July 2020; revised May 2021; accepted June 2021