Scalable Spatio-temporal Top-k Interaction Queries on Dynamic Communities

ABDULAZIZ ALMASLUKH, College of Computer and Information Sciences, King Saud University, Riyadh, Saudi Arabia
YONGYI LIU, Department of Computer Science and Engineering, Center for Geospatial Sciences, University of California, Riverside, United States
AMR MAGDY, Department of Computer Science and Engineering, Center for Geospatial Sciences, University of California, Riverside, United States

Social media platforms generate massive amounts of data that reveal valuable insights about users and communities at large. Existing techniques have not fully exploited such data to help practitioners perform a deep analysis of large online communities. Lack of scalability hinders analyzing communities of large sizes and requires tremendous system resources and unacceptable runtime. This article proposes a new analytical query that identifies the top-*k* posts that a given user community has interacted with during a specific time interval and within a spatial range. We propose a novel indexing framework that captures the interactions of users and communities to provide a low query latency. Moreover, we propose exact and approximate algorithms to process the query efficiently and utilize the index content to prune the search space. The extensive experimental evaluation on real data has shown the superiority of our techniques and their scalability to support large online communities.

CCS Concepts: • **Information systems → Information retrieval query processing;**

Additional Key Words and Phrases: Community, query processing, users interactions, spatio-temporal query

ACM Reference Format:
Abdulaziz Almaslukh, Yongyi Liu, and Amr Magdy. 2024. Scalable Spatio-temporal Top-k Interaction Queries on Dynamic Communities. *ACM Trans. Spatial Algorithms Syst.* 10, 1, Article 6 (March 2024), 25 pages. https://doi.org/10.1145/3648374

1 INTRODUCTION

Online communities have become more popular with the advancement of user-generated data platforms. They are rich with useful and important data. Users are naturally forming communities similar to the communities in the real world based on common interests, ideas, and locations. These communities can be extremely useful in rescue missions [18, 39, 54], disease prevention [19], urban planning [45], and public safety [11, 37]. Additionally, the research community has paid significant attention to detecting [43] and searching communities [30] that are homogeneous and

The work started while A. Almaslukh was at the University of California, Riverside.
Authors' addresses: A. Almaslukh, College of Computer and Information Sciences, King Saud University, Riyadh 11362, Saudi Arabia; e-mail: aalmaslukh@ksu.edu.sa; Y. Liu and A. Magdy, Department of Computer Science and Engineering, University of California, Riverside, CA 92521, USA; e-mails: yliu786@ucr.edu, amr@cs.ucr.edu.

ACM 2374-0353/2024/03-ART6
https://doi.org/10.1145/3648374

have shared characteristics in common. Nevertheless, analyzing a single community has received very little attention. For example, a social scientist might analyze the US teenagers on Facebook (representing 5.2 millions users [26]) or on Instagram (representing 57 millions users [40, 41]) to find out what they interacted with during the past week on different topics such as bullying, youth suicide, and COVID-19 pandemic. This kind of analysis is very crucial in understanding, making the right decisions and offering better services to the target community. In fact, there are hundreds of online communities with multi-million users [4], each of them has lots of large sub-communities, and it is recognized that analyzing these communities is of high importance in various applications [13, 31, 32, 53, 59, 60, 62]. However, existing querying techniques are not scalable to perform such analysis on large online communities that involve millions of users.

Queries on online communities come in various forms, e.g., finding top-k influential users [5, 14, 70, 73] and top-k experts in knowledge communities [74]. However, direct queries such as *"finding top-k posts on which teenagers in the US interacted the most during the past day"* are not currently supported on large communities. These queries are useful in studying various types of online communities during different events, such as underrepresented communities during political events or pandemic responses. To support this query using existing techniques, we need to traverse millions of users to filter out their interactions temporally and based on the context of the posts. This requires very expensive computations and unacceptable query latency due to the huge number of users. Therefore, existing techniques are limited to reveal the full potential of online user-generated data.

In this article, we propose a spatio-temporal community query that finds top-k posts with which the given community has interacted during a given time and within a given spatial range. The nature of the community is dynamic over time; some individuals join and others leave. Thus, this work mitigates this problem to help the query capturing the dynamism of the community with the support of the user index to process the query more efficiently when the community is evolving. The proposed query helps practitioners to understand the interests of different communities over different temporal and spatial ranges. However, processing this query using traditional indexing techniques, e.g., classical RDBMS, requires a tremendous amount of system resources and CPU time. In specific, there are two main challenges: (1)The users' interactions with the posts are very huge in number and constantly increasing. For instance, Facebook users upload on average 240k photos/min and generate 4M likes/min [25]. Therefore, digesting this information is a major challenge to reduce the query latency and minimize the system resources overhead. (2)The community nature is dynamic over time; so users join or leave communities all the time. Thus, the system must account for this dynamism to support evolving communities in practice.

To address these challenges, we devise novel scalable indexing and query processing that deals with communities as whole units instead of processing individual users' data. Our index efficiently supports established communities to process efficient queries on them with limited system resources. Established communities are formed by users or system administrators based on arbitrary criteria that connect all community users, e.g., users of common interest, geographic location, or demographic groups. The proposed index shares the same spirit of database indexes where admins create and drop indexes on demand on any user-defined community that is frequently queried to tune the system performance. Every community interaction is being indexed temporally through time slicing with temporal resolution T. Each community is predefined by a set of users, and every user interaction is indexed to the corresponding community index. Whenever a user interaction occurs, it is indexed to the most recent time slice to all communities to which she belongs. The proposed index stores aggregated information that reflects the total interactions of the community with various posts. However, the community index does not index any interactions from non-member users. Therefore, dynamic communities, where users join and leave communities

over time, have no pre-built index in the system. To support community dynamism, we complement the community index with a new indexing component to index all users' interactions based on time slices that correspond to the community index slicing. This component is utilized during query processing to handle newly joined or left users, which seamlessly supports community dynamism. Combining the two indexing components, community index and user index, makes it easy to avoid expensive restructuring for the community index frequently and enables efficient support for large and dynamic communities.

The query processor smartly exploits the index content to provide low query latency. We propose three query processing techniques, two exact algorithms, *baseline ComCQ* and *fast ComCQ*, and one approximate algorithm, *approximate ComCQ*. The query processor locates parts of both community and user indexes that intersect with different query predicates, temporal, spatial, and user predicates. Then, corresponding index contents are aggregated to find top-k posts for the community. Depending on how large the query predicates, processing the index aggregates could still be highly expensive. Thus, the approximate query processing techniques smartly select the most important content to process. This enables querying large communities and extended spatial areas and temporal ranges using limited system resources.

This work is a significant extension for our work in Reference [9] to enable data practitioners to analyze dynamic online communities where users join and leave over time. To support community dynamism, we significantly extend both indexing and query processing to facilitate joining and leaving communities for users at scale while still providing efficient performance. For indexing, we added a new user index, in addition to the community index. The user index organizes aggregate data from all individual users based on temporal slices. Then, it is used to compute a community delta component that represents newly joined or left users to be used in providing accurate query answers on the changing community. For query processing, we added a new approximate query processing technique, *approximate ComCQ*, that supports efficient aggregation on large communities. Instead of aggregating all data in main-memory, *approximate ComCQ* smartly selects the most important parts of the data to enable efficient queries on large highly dynamic communities that encounter large numbers of joining or leaving users.

The extensive experimental evaluation of our proposed techniques on real data has shown the efficiency of our indexing framework and query processing techniques. Our contributions are summarized as follows:

- We introduce a new analytical query over large online communities that returns the top-k posts that a community has interacted with.
- We propose a novel indexing paradigm with multiple components to efficiently support both established and dynamic communities.
- We develop different query processing techniques to efficiently process the query even when the system resources are limited.
- We provide an extensive experimental evaluation on real data to show the superiority of our proposed techniques.

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 community and users indexing and query processing techniques, respectively. Section 7 provides an extensive experimental evaluation. Section 8 concludes the article.

2 RELATED WORK

Existing related work mostly focuses on community search and detection, in addition to other problems. There is no current research work that addresses efficient querying for large online

communities, to the best of our knowledge. This section covers notable works that address the community problems in different contexts.

Community Search and Detection. The community search problem was first proposed in Reference [68]. The problem refers to finding a sub-graph of cohesive community with respect to a set of query nodes such as users. Different community search models have been proposed, including k-core [21, 68], k-truss[38], k-clique [20], location-aware community search [48], and degree-based label propagation algorithm for detecting community search [24]. Some techniques focus on improving the quality of the cohesion [29] or output accuracy [52] on attributed graphs, while other techniques introduce a new dimension such as spatial-aware community search [16, 28]. We refer the reader to this survey for a detailed evaluation of the existing techniques [30].

However, finding communities in a large graph is a fundamental and well-studied problem in the data-mining domain known as community detection. In contrast to community search, community detection tries to find all the communities in a graph rather than a single specific community. There are many proposed techniques for community detection [7, 23, 57, 72, 77]. Some techniques focus on optimizing the modularity measure, which is widely used for the goodness of the community [17]. Other techniques work on scalability [51] and provide search strategies that are faster than the traditional modularity-based search. Comprehensive surveys on community detection techniques are presented in References [34, 44, 58].

Our work is orthogonal from, yet complementary to, these works. In fact, the output communities in this work are valid as input to our querying module. Thus, we enable efficient querying on large communities that is not addressed by these works.

Graph Data Management. With the rapid advancement of social networks and their applications, researchers have paid tremendous attention to big graph data management systems [1, 50, 67, 75] in terms of modeling [46], storing [71], accessing [6], and querying [10, 61] graph data. We refer the reader to this survey for detailed approaches and techniques on graph data management [42]. Although graph data management systems are very suitable to be adopted by many community problems due to the flexible representation of these data, they are not very efficient for community-centric queries. Graph-based techniques are well-suited for users-centric queries such as finding top-k users who are expert in certain subjects based on the given user social network. However, community-based queries are more complex and require aggregated results and auxiliary indexes to capture and maintain the structure of a community to respond efficiently to the queries, especially for the large-size communities where the graph-based techniques do not scale due to the inherently query complexity. Moreover, these systems process communities as a set of individual users, leading to unacceptable query latency on a large number of users. Our work addresses this limitation by introducing community-centric data management structures that deal with a community as a whole unit rather than a bag of users. This enables efficient data management and provides a low query latency.

Other Community Problems. Several scattered problems have been studied on online communities, such as community-based question answering [56, 69] and multi-dimensional communities [33]. Our work is orthogonal from all these works, focusing on identifying the top interactions within a large community of users over spatial and temporal dimensions.

Top-k Spatial Queries. There has been a variety of research work that focused on spatial queries with different predicates, including top-k spatial keyword queries [15, 63], top-k spatial temporal queries [35, 49], and top-k spatial temporal keyword queries [22, 36, 47]. In the context of user interactions, interactive top-k spatial keyword query [78] has been proposed by learning user preferences at each round of interaction to improve the quality of top-k results. However, these works are inherently user-centric queries, while this work focuses on the community as a whole.

3 PROBLEM DEFINITION

Before we define the problem formally, we first define the terminologies used throughout this work:

- Post: is a piece of content that is generated by a user and usually associated with text, timestamp, and geo-location.
- Interaction: is an event triggered by the users such as liking, commenting, or sharing posts.
- Established Community: is a community that has been around for a long time with a fixed users base.
- Dynamic Community: is a community that is constantly changing and evolving where new users join and existing users leave.
- Ad hoc Community: is a community that is formed to respond to an urgent event or formed for a specific purpose.
- User Index: is a data structure that stores information about users' interactions partitioned into subsets based on the time intervals.
- Community Index: is a data structure that stores aggregated information about community interactions partitioned into subsets based on the time intervals.

The existing work on community detection and search is orthogonal and could serve as a preprocessing phase to generate our established communities. However, our work focuses on scalable indexing and processing complex queries on such detected communities, especially when they involve a substantial number of users. A community could be a social media group where individuals can join and leave or could be as simple as followers of a public figure or a company. In the former case, an administrator of the group, which is usually appointed upon the group creation, manages the enrollment process or the group could be open where any individual can join and leave anytime. In the latter case, the users choose to follow or unfollow, which is the common practice on all major social media platforms such as Facebook [2] and LinkedIn [3].

We evaluate community queries on a dataset D that consists of posts P generated by the users. Each post $p \in P$ is represented with four main attributes (oid, kw, $timestamp$, $location$), where oid is a unique identifier of the object, kw is a set of keywords that represent the textual description of the object, $timestamp$ is the time when the object is posted, and $location$ is a tuple that represents the geographical location, i.e., latitude and longitude coordinates, of the object when it is posted. Table 1 shows a sample of posts. In this work, we use the terms post and object interchangeably. A *virtual community (C)* is defined as a set of users $\{u_1, u_2, u_3, \ldots, u_n\}$ as members of the community where $|C| = n$ is the community size. An *interaction* is a specific action that a user u performs on a post p, such as like, reply, or share. Table 2 shows a sample of interactions between the users and the posts with different type of interactions. We formally define our community query as follows:

Community Spatio-Temporal Interaction Query (*CSTIQ*): given a virtual community C , a time interval T_q=[$t1$,$t2$], an integer k, an optional point location c and a spatial range r, and an optional set of keywords W, CSTIQ query finds a set of k posts P_o so each post $p \in P_o$ satisfies the following: (1) $p.location$ lies within a spatial range centered at c with radius r, (2) $p.kw \cap W \neq \emptyset$, i.e., p contains one or more of the query keywords, and (3) p is ranked top with respect to a ranking function $F(C, p, T_q)$ that is defined as follows:

$$F(C, p, T_q) = \sum_{\forall u \in C} Interaction(u, p, T_q),$$

where $Interaction(u, p, T_q)$ is the total number of interactions that a user u makes with a post p during a time interval T_q. F ranks posts based on total interaction from the given community C, so the query outputs the top-k posts that have been most popular in C during the query time

Table 1. Sample of Objects

OID	Keywords	Timestamp	Object Location
$o1$	NBA, Lakers, Kobe, Bryant	05-05-2022 01:20:58	34.05° N, 118.24° W
$o2$	Coronavirus, Wuhan, China	05-05-2022 03:17:27	37.77° N, 122.41° W
$o3$	Election, GOP, Caucus	05-05-2022 05:55:23	40.71° N, 74.00° W
$o4$	Stocks, Tesla, Electric, Cars	05-05-2022 05:56:19	33.98° N, 117.37° W
$o5$	Windy, Weather, Wildfires	05-05-2022 15:46:17	32.77° N, 96.79° W
$o6$	Tokyo, Olympics	05-05-2022 18:06:14	35.67° N, 139.65° E
$o7$	Beaches, Summer, Surfing	05-05-2022 19:17:09	25.76° N, 80.19° W
$o8$	Sunny, Sport, Marathon	05-05-2022 19:33:06	42.36° N, 71.05° W

Table 2. Sample of Users' Interactions with Objects

UID	CID	OID	Location	Action	Timestamp
$u1$	$c1$	$o1$	34.05, −118.24	Like	5-5-2022 20:18:50
$u2$	$c1, c3$	$o2$	37.77, −122.41	Reply	5-5-2022 20:18:27
$u3$	−	$o3$	40.71, −74.00	Share	5-5-2021 20:18:23
$u1$	$c1$	$o3$	40.71, −74.00	Like	5-5-2022 20:18:19
$u4$	$c1$	$o1$	34.05, −118.24	Like	5-5-2022 20:18:17
$u2$	$c3$	$o5$	32.77, −96.79	Reply	5-5-2022 20:18:14
$u5$	−	$o4$	33.98, −117.37	Mention	5-5-2022 20:18:09
$u6$	$c2$	$o1$	34.05, −118.24	Like	5-5-2022 20:18:06

period. Both keywords and locations are optional, which enables the query to be flexible in different aspects. First, providing keywords W enables the query to search for popular posts related to a specific topic, while omitting W allows discovering any popular topics within the community C. Second, providing or omitting a query location allows flexibility to pose either pure spatio-temporal or temporal queries. This is important due to several reasons. It facilitates supporting virtual communities that are not tied to a physical location, e.g., data-mining research community, and it enables to query global-scale communities, e.g., animal rights activists.

4 CSTIQ INDEXING

Indexing user interactions in large online communities faces two main challenges. First, user interactions are huge in number, up to an order of magnitude larger than the posts. For instance, our evaluation real dataset has 80 million posts with over 1.9 billion interactions. With the high arrival rates of such data, the index needs a light and efficient digestion mechanism. Second, the community nature is dynamic over time, so users join and leave. The index should avoid expensive rebuilding and restructuring operations.

To address these challenges, while serving our query as defined in Section 3, we introduce ***Community Spatio-Temporal Indexing Query (CSTIQ)*** framework. Figure 1 shows an overview of *CSTIQ* framework. It consists of multiple indexing components. The first component is the *community index* that indexes interactions in established communities where the community users are already known in advance. This index aggregates the interactions of all users within the community. Thus, it will expedite the query processing by dealing with the whole community as one unit instead of millions of individual users. This index is light and temporally sliced to handle an excessive volume of interactions efficiently. However, the community index helps the

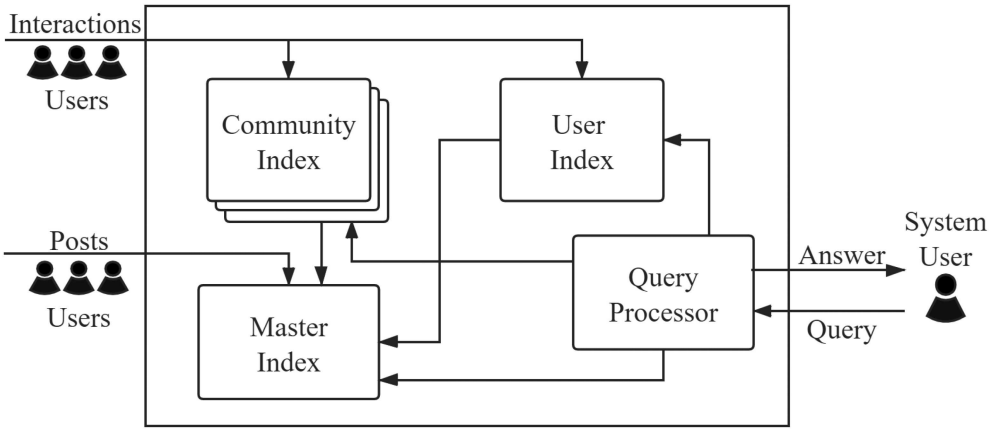


Fig. 1. CSTIQ framework.

query processor only when querying an established community. Thus, this index cannot serve any query that targets either a new community or even a slight variant of an existing community. To support dynamic communities, CSTIQ introduces the second component, the *user index*, to index all users' interactions regardless of their community memberships. The user index stores any user interactions aggregated by user IDs. This index stores more detailed information than the community index and can serve any query, including the ad hoc communities that have never been seen before. The two indexing components store only post ids and locations. These ids are used to retrieve data from a third indexing component, called the *master index*, that stores all available information about posts, e.g., user profile or keywords.

CSTIQ index always includes one user index, but it might include more than one community index. The proposed community index is created and dropped on demand to tune the system performance for frequently queried communities, just like database indexes where admins create and drop indexes on demand to tune performance. CSTIQ index organizes data based on temporal and keyword dimensions while storing raw location information, enabling the query processor to serve CSTIQ query with all its three-dimensional parameters, time, location, and keywords. Both user and community index have two sub-components: an in-disk component and an in-memory buffer to swap the data between memory and disk. The buffer adopts the famous **least recently used (LRU)** policy to evict the old data. The query processor blends data from different indexing components to provide low query latency, as detailed in the following sections. The rest of this section details both the community index and user index. The pseudocode for the index update is presented in Algorithm 1.

5 QUERY PROCESSING

5.1 Community Index

This section presents the community index. This index summarizes interactions of an established community as one whole unit. The section details index structure and update operations.

Index Structure. The community index has two identical indexing components: memory-based index and disk-based index. The former is mainly for the most recent community interactions to enable light and efficient digestion for excessive recent data. The latter is for the relatively old interactions that are evicted from the main memory and usually receive much fewer updates. Both components are divided into non-overlapping time slices $[T_0, T_1, T_2, \ldots, T_{now}]$. These time slices are equal in size and of user-defined length. They can be set to cover any period of time, e.g., 24 hrs,

ALGORITHM 1: CSTIQ Index Update

```
Input: Posts (P), Interaction (IR)
for each p ∈ P do
    Update Master Index:
    Index p in the Master Index
for each (u,p) ∈ IR (user u interacts with post p) do
    Update User Index:
    if u ∈ H_u.key then
        H_u[u].append(p.oid, p.location)
    else
        H_u[u] = [(p.oid, p.location)]
    Update Community Index:
    CI = the community index of the community u belongs to
    H_m = master hash structure in CI
    if p.oid ∈ H_m[ID] then
        H_m.ID[p.oid][Count] = H_m.ID[p.oid][Count] + 1
    else
        H_m.ID[p.oid][Count] = 1
    I = the inverted index in CI
    for each keyword kwd ∈ p.kw do
        if kwd ∈ I then
            H_i = the hash structure associated with kwd
            H_i[p.oid][Count] = H_i[p.oid][Count] + 1
        else
            add kwd to I and create H_i associated with kwd
            H_i[p.oid][Count] = 1
Return Updated Master Index, Community Index, User Index
```

12 hrs, or 1 hr, based on available system resources and data size. Whenever the current time slice has passed, a new time slice is created and marked to be T_{now}. All interactions happening within the current time interval are indexed into the T_{now} slice. Each time slice includes two data structures: (1) a master hash structure (H_m), where the post ID is a key and the value is the post location and total interactions; (2) an inverted index (I), where every entry represents a keyword, and each keyword points to a hash structure (H_i) similar to the master hash structure. Any query that does not have a keyword, H_m data structure will be utilized. However, the query that specifies keyword parameter, I will be used to process the query efficiently. The Community Index is constructed and updated incrementally. Initially. both the master hash structure H_m and the inverted index are empty. As interactions occur between users and posts, these changes are reflected within the community index, as discussed as follows:

Figure 2 shows the structure of an example community index. The figure shows three time slices, labeled as T_0, T_1, and T_n. T_0 has H_m, which stores object ids, locations, and the counts of interaction from this community such as $o1$ has 66 community interactions and location coordinates (30.26, −97.74). In addition, T_0 has I, which stores *Pizza* keyword, which points to H_i that has two posts, $o3$ and $o4$, and their count values are 40 and 15 community interactions and locations (41.87, −87.62) and (33.98, −117.37), respectively.

Index Update. Whenever a user who belongs to a community C interacts with an object *oid*, C's corresponding index is updated accordingly. The in-memory index updates two data structures:

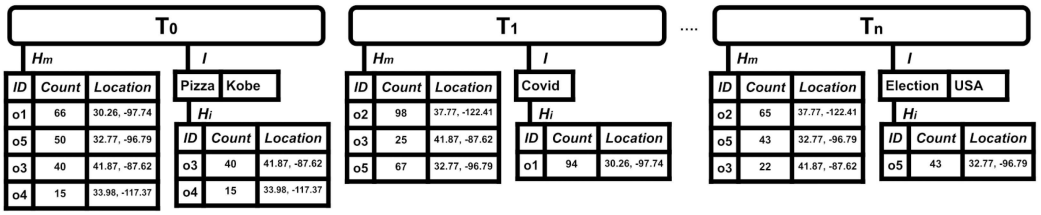


Fig. 2. Community index structure.

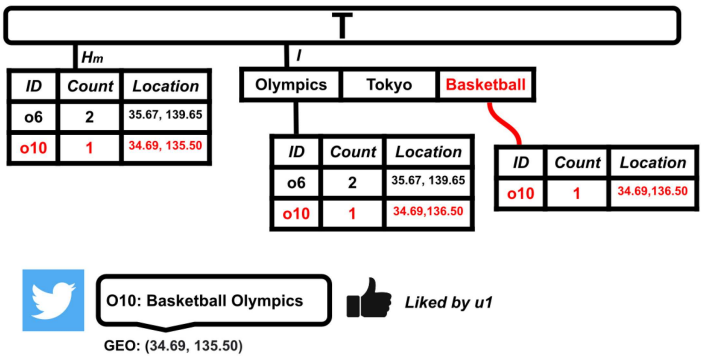


Fig. 3. Community index update example.

the master hash index H_m and the inverted index I. First, *oid* is looked up in H_m. If *oid* is found, then its count is incremented by one; otherwise, it is inserted with count set to one and location set to *oid.location*, marking the first interaction from C's users to *oid*. Then, the inverted index I will index *oid*'s keywords. For a specific post p, $p.kw$ is a set of keywords that represent the textual description of the post. We extract each keyword from $p.kw$ and apply necessary preprocessing techniques. In the inverted index I, every keyword kw_i is also associated with a hash structure H_i, which is similar to H_m. During the processing of a keyword kw, if kw is not already present in I, then it gets added to I and an associated hash structure for it is created. Conversely, if I does include kw, then the relevant hash structure linked to kw is identified and the corresponding count value is subsequently updated.

Figure 3 shows an example of updating the community index at temporal slice T. Consider a user from the community, denoted as u_1, who interacts with a post o_{10}, which is characterized by the textual description "Basketball Olympics" with a geographical location 34.69, 135.50. Both the indices H_m and H_i associated with the terms "Olympics" and "Basketball" are to be updated in response to this interaction. Since o_{10} is not in the ID column of H_m, a new entry with ID o_{10} and an initialized count of 1, together with the geographical location of the post, is added to H_m. Further, the keywords from o_{10}, i.e., "Olympics" and "Basketball" are extracted and used to update I. Since "Olympics" is already in I, the corresponding H_i is retrieved and a new entry is added at the bottom. The word "Basketball" is not in I, so it is added to I and a new H_i associated with "Basketball" is created and initialized.

Once the current time slice expires, both H_m and I are concluded, and a new time slice is initiated with empty structures. H_m and each H_i of the concluded time slice are sorted based on the number of interactions in descending order. The sorted time slices enable efficient query processing, as detailed in the following sections. After the total designated memory budget is consumed, the least recently used time slices are evicted from the in-main index to the corresponding in-disk index.

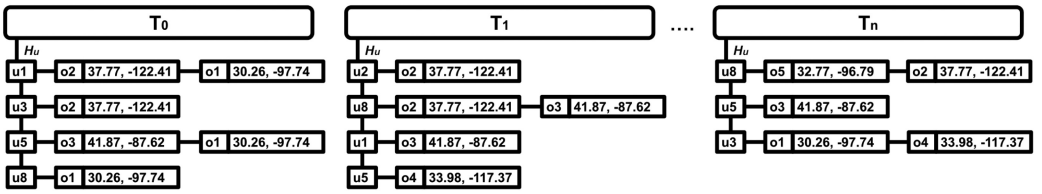


Fig. 4. User index structure.

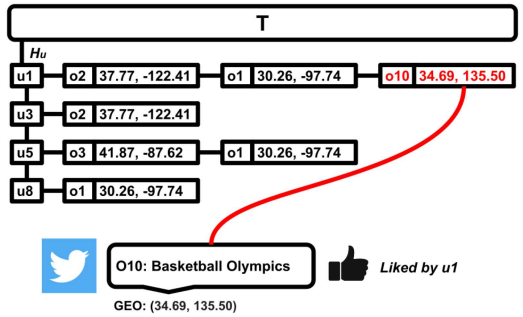


Fig. 5. User index update example.

5.2 User Index

The user index organizes all users' interactions regardless of their affiliations to communities. This enables the query processor to efficiently support community dynamism as users join and leave established communities. The index has two identical components, an in-memory index and an in-disk index. Each of them is divided into disjoint time slices of length T, which are corresponding to the ones of the community index. To efficiently index the huge number of interactions, a hash structure H_u is used where the user ID serves as a key and the value is a list of posts that includes ids and locations that the users interacted with. This hash structure H_u is light and provides a very effective search and insertion. When the time slice T_0 is expired, a new time slice T_1 is created. After consuming all designated memory budgets, the least recently used time slices are evicted from the main memory and stored in the disk-based index to free up space for new time slices in memory.

Figure 4 shows the user index structure. The figure shows three time slices, T_0, T_1, and T_n. T_0 points to a hash structure H_u that stores user ids $u1$, $u3$, $u5$, and $u8$. $u1$, for example, points to a list of objects with ids $o2$ and $o1$ and locations $(37.77, -122.41)$ and $(30.26, -97.74)$, respectively, with which $u1$ has interacted during T_0.

Figure 5 illustrates the update of the user index. In this case, the user u_1 interacts with the post o_{10}; consequently, an entry $(o_{10}, (34.69, 135.50))$ is added to end of the list corresponding to u_1 in the user index.

This section describes the query processing of *CSTIQ* query, which is defined in Section 3 utilizing the indexing framework that is introduced in Section 4. The query processor employs different techniques to minimize the overhead on system resources and provide efficient query latency. More specifically, we propose three algorithms to process *CSTIQ* query, namely, *baseline ComCQ*, *fast ComCQ*, and *approximate ComCQ* algorithms. The rest of this section introduces a high-level query processing framework that is utilized for the three algorithms, and then we detail the specifics for each algorithm.

5.3 Query Processing Framework

This section introduces a three-step query processing framework. The first two steps are shared among all our proposed algorithms, while the third step differs based on the algorithm. Every *CSTIQ* query takes a community C parameter as an input, and there are three different cases for the community C parameter: **Case 1**: C has been previously established and an index for C has been already built and being maintained. **Case 2**: C is not established but it is a variation of an established community C_e, where C=$C_e \pm U_\Delta$. Thus, the list of user changes U_Δ must be considered in the query processing. **Case 3**: Neither C has been established nor the index for C has been built. Our query processor consists of three steps:

(1) Step 1: Index retrieval. Given a community C, as an input of *CSTIQ* query, in this step, we check which case the community C falls into. If C falls into **Case 1**, then the index of C is retrieved and fed to Step 2. If C falls into **Case 2**, then another auxiliary community in-memory index is being built utilizing the index of users interactions for only U_Δ users within the given query time interval; we call this auxiliary community index U_Δ *index*. The index for C_e and the newly generated auxiliary U_Δ index are retrieved and fed to Step 2. If C falls into **Case 3**, then a new index is being built utilizing the index of users interactions for community C within the given query time intervals. Then, the new index is fed to Step 2.

(2) Step 2: Temporal and keyword filtering. Given the community C index from Step 1, the query processor retrieves the time slices that overlap with the query time interval T=[$t1$,$t2$] and stores a copy of each slice in in-memory list L_j that corresponds to interval j. A list L_j stores the entries of its time slice in descending order of their total interactions. These entries are already ordered as part of the indexing process, so it adds no sorting overhead. The exception to this is the most recent slice T_{now} that is not ordered, so it is copied and the corresponding list L_{now} is sorted on the spot if it lies within the query time. If the query has keywords, then the query processor retrieves H_i hash structures that are corresponding to every query keyword and merges them in one list L_j ordered by total interactions. The merged list holds the union of the posts and the total interactions of each post. If the query does not have any keywords, then the query processor just copies the master hash structure H_m. To reduce the overhead of reading back-and-forth from the disk, all lists are stored in an in-memory buffer. When the in-memory buffer is full, the **least recently used (LRU)** policy is adopted to evict data to continue serving incoming queries.

If U_Δ index exists, then the copied lists are updated to reflect U_Δ's users changes, whether adding their interactions or subtracting them. For each time slice in U_Δ index, we iterate over all posts and update corresponding list L_j. Finally, the updated list is reordered based on the new total interactions. The updates affect only the copied lists in the main-memory. Thus, the original copy of community time slices, either in memory or in disk, is not affected. Finally, the lists L_j are fed to Step 3.

(3) Step 3: Spatio-temporal aggregation. Given the lists of posts that are retrieved in Step 2, these lists are processed to return top-k posts that the community C interacted with during the query time interval and within the query spatial range. Due to variations in query parameters, data sizes, and system resources, we have three different variations of our query processing technique that accommodate different needs. Each of these variations perform Step 3 differently. The following sections detail each of them.

5.4 Baseline ComCQ

In case of small data sizes, we provide a baseline algorithm, called **baseline ComCQ (*B-ComCQ*)**, that performs straightforward aggregation to process *CSTIQ* query and returns exact results. *B-ComCQ*'s input is the lists L_j that are retrieved in Step 2. *B-ComCQ* creates a new hash structure

ALGORITHM 2: Baseline ComCQ

```
Input: A CSTIQ query q
Initialization:
Retrieve the community index C and construct its variation based on the user index if needed
Filter the entries from each time slice and obtain the filtered lists of entries L = [L_1, ..., L_m]
Top-k Aggregation:
H_a = { }
for each L_i ∈ L do
    for each entry e = (id, count, loc) ∈ L_i do
        if distance(loc, q.c) < q.r then
            if id ∈ H_a.keys then
                H_a[id] = H_a[id] + 1
            else
                H_a[id] = 1
ans = rank the top-q.k entries from H_a according the count
Return ans
```

H_a that is similar in structure to H_m and H_i. The purpose of H_a is aggregating the total of interactions for each post. For each list L_j, *B-ComCQ* iterates over all entries in its corresponding hash structure. For each entry, it checks if the post p lies within the query q's spatial range, i.e., $distance(p.location, q.c) < q.r$. If so, then its total interactions are added to H_a. After processing all lists, H_a has all the aggregated entries. Finally, *B-ComCQ* sorts H_a, and the top-k entries of the sorted H_a are returned as the final answer. Algorithm 2 gives the pseudocode for Baseline ComCQ.

5.5 Fast ComCQ

B-ComCQ performs exhaustive search to find the top-k posts. This is inefficient when the number of time slices or the number of entries is even moderate. Therefore, we develop an efficient, yet exact, algorithm that is inspired by Fagin's TA algorithm [27] called ***fast ComCQ* (*F-ComCQ*)**. *F-ComCQ* assumes that all time slices can fit in main memory. *F-ComCQ* does not need to access every entry in every list L_j. Instead, it smartly prunes entries that surely have no chance to be in the top-k posts. Thus, this will save many unnecessary searches. For each list L_j that is fed from Step 2, *F-ComCQ* performs five steps:

(a) **Initialization with spatial filtering**. *F-ComCQ* iterates items of L_j in order until the first item that lies within the query spatial range is found. This item is then inserted into a priority queue Q with priority score equal to the number of interactions of the item. This repeats to every list L_j, so Q is initialized with the most popular post from each time slice. An ordered list *Ans* is initialized to keep track of the top-k items found so far in every iteration. A variable *SumQ* is initialized with the sum of priority scores in Q.

(b) **Top item pickup with spatial filtering**. If the priority queue is not empty, then remove the top entry e_Q of the queue Q and insert into Q the next entry that lies within the query spatial range from the same time slice of e_Q and update *SumQ* accordingly. We maintain a pointer in each list L_j that always points to the first entry that has not been visited so far to facilitate iterating over items of the same time slice.

(c) **Temporal aggregation**. Using the hash structures H_m and H_i of each time slice, this step calculates the total interactions of e_Q in all time slices. We check if e_Q exists in H_m or H_i, and we add its interactions to the summation variable. After iterating over all hash structures, the summation variable includes the total interactions of e_Q in all time slices.

ALGORITHM 3: Fast ComCQ

```
Input: A CSTIQ query q
Initialization:
Retrieve the community index C and construct its variation based on the user index if needed
Filter the entries from each time slice and obtain the filtered lists of entries L = [L_1, ..., L_m]
Top-k Aggregation:
pq = ( ) //priority queue
for each L_i ∈ L do
    for each e = (id, count, loc) ∈ L_i do
        if distance(loc, q.c) < q.r then
            pq.add(e)
            break
ans = ( )
sumQ = sum of priority scores in pq
while pq is not empty do
    e = pq.dequeue()
    (id, count, loc) = e
    L = the list e comes from
    add the next entry that satsifies the distance constraint to pq from L
    update sumQ
    count_total = e.count+ interactions with this post from other time slices
    if ans.size < k then
        put (id, count_total) in ans
    else
        if count_total > kth score from pq then
            remove the kth item in ans
            put (id, count_total) in ans
    if ans > k AND kth score in ans < sumQ then
        break
Return ans
```

(d) **Top-k answer update**. If size of *Ans* list < k, then insert e_Q into the ordered list *Ans*. Once the size of *Ans* grows to k and larger, we maintain *lower_bound* as the kth, i.e., lowest, score in *Ans*. If total interactions of e_Q > *lower_bound*, then we remove the kth item in *Ans* and insert e_Q in order, otherwise, e_Q is discarded. Then *lower_bound* is updated with the new lowest score in *Ans*.

(e) **Search termination**. If *Ans* size ≥ k and *lower_bound* ≥ *SumQ*, then the search stops and *Ans* is the final answer. Otherwise, we repeat steps b through d.

Picking up top popular items first and termination based on existing *Ans* scores eliminate any unnecessary processing and speed up the search significantly, as shown in our experiments. Algorithm 3 gives the pseudocode for Fast ComCQ.

5.6 Approximate ComCQ

F-ComCQ assumes that data of all query time slices fit in main memory at once. When this assumption is invalid, meaning that the available memory is not enough or the query time or data size are too large to fit in main memory, we propose an approximate version of *F-ComCQ*, called ***approximate ComCQ (A-ComCQ)***, which utilizes the available memory budget to provide highly accurate query answer. *A-ComCQ* exploits the observation that users' interactions obey

the power-law distribution [8, 12, 55, 64–66, 76], so most of the interactions come from a small number of users. Thus, loading only part of the data and ignoring the long tail still achieves very high accuracy and reduces the computation overhead tremendously. Accordingly, *A-ComCQ* retrieves only top items in every time slice.

A-ComCQ takes an input parameter α that represents the number of top items to retrieve from each time slice. When $\alpha \rightarrow \infty$, all the time slice data is loaded to memory and *A-ComCQ* acts similar to *F-ComCQ*. With smaller values of α, less data is loaded to memory to adjust the query processor for the limited system resources. For each time slice, only α top items are loaded from disk and copied to the lists L_j that corresponds to time slice j. Then, all the five steps of *F-ComCQ* are applied on the loaded lists. As shown in our experiments, for practical values of query parameters, the output accuracy of *F-ComCQ* and *A-ComCQ* are similar with very marginal inaccuracy, as some items are missed due to data truncation. However, the query processing time is decreased significantly, since a huge portion of data has been truncated.

6 COMPLEXITY ANALYSIS

This section presents time and space complexity analysis for the proposed indexing and query processing algorithms.

6.1 Complexity of Indexing

We discuss the time and space complexity of indexing for both the user index and the community index for a specific community C. Denote the maximum length of a post, i.e., the maximum number of words from a post, as L_{max}, the total number of interactions as N, and the number of temporal slices as T.

6.1.1 Complexity of Community Index. In this section, we discuss the time and space complexity of the community index proposed in Section 5.1.

Time Complexity: Consider an interaction generated by community C, associated with a post p. Identifying the presence of p in H_m requires $O(1)$ time. Further, updating H_m, i.e., increasing the count or creating a new entry, also takes time $O(1)$. Similarly, for the keywords contained in $p.kw$, identifying the presence of the corresponding H_i index incurs a time complexity of $O(L_{max})$ due to a maximum of L_{max} keywords from a post. Similar to updating H_m, updating each involved H_i takes time $O(1)$. The cumulative time taken to update the H_i indices associated with all the keywords of a post amounts to $O(L_{max})$. Therefore, the time complexity of indexing an interaction takes $O(1) + O(L_{max}) = O(L_{max})$. Since there are N interactions, the time complexity of building the Community Index is $O(L_{max}N)$. Although hash collisions and the need for resizing may occur as the size of the index increases, the amortized time complexity for accessing the hash structure consistently remains at $O(1)$ per interaction.

Space Complexity: For an interaction generated by community C, associated with a post p. If p is encountered for the first time, then we record in H_m the attributes $p.ID$, $p.count$, and $p.location$, all of which require $O(1)$ storage. Also, we record the same attributes in each H_i index corresponding to different keywords in p, which takes $O(L_{max})$ storage, since p has at most L_{max} keywords. In cases where p has been previously observed, no additional space is required in either H_m or H_i indices, since only an update to the interaction count is needed. Given N interactions, the maximum number of distinct posts is $O(N)$. Consequently, the upper bound for storage in H_m and H_i is $O(N)$, and $O(L_{max}N)$, respectively, which results in $O(L_{max}N)$ space complexity to build the Community Index.

6.1.2 Complexity of User Index. In this section, we discuss the time and space complexity of the user index proposed in Section 5.2.

Time Complexity: When a user u makes an interaction with a post p, we utilize H_u, with the user ID as the key, to identify the list of posts with which this user has previously interacted. This process takes at an amortized time complexity of $O(1)$, as explained above. Subsequently, the post involved in the current interaction is appended to the end of this list, a procedure also requiring $O(1)$ time. Given that there are N total interactions, the overall time complexity of building the User Index is $O(N)$.

Space Complexity: Every time a user u makes an interaction with a post p, the ID of post p along with its geographical coordinate is stored in the corresponding list, which takes $O(1)$ storage. Since there are N interactions, the total space complexity is $O(N)$.

6.2 Complexity of Query Processing

We discuss the time and space complexity of running different query processing algorithms on a specific community index. Similar to the above, we assume the query spans T temporal slices, involves N interactions, and returns the top-k posts.

6.2.1 Complexity of Baseline ComCQ. This section gives the time and space complexity of the Baseline ComCQ algorithm proposed in Section 5.4.

Time Complexity: Baseline ComCQ aggregates interactions among different temporal slices, which ends up having $O(N)$ posts in the hash structure, taking $O(N)$ time. Sorting and identifying the top-k entries takes $O(NlogN)$ time, which gives a time complexity of $O(N + NlogN) = O(NlogN)$. Consequently, the time complexity of Baseline ComCQ is $O(NlogN)$.

Space Complexity: Baseline ComCQ aggregates post interactions using hash structure H_a, taking space $O(N)$ for up to N posts, which leads to a space complexity of $O(N)$. Sorting and identifying the top-k entries takes also $O(N)$. Consequently, the space complexity of Baseline ComCQ is $O(N)$.

6.2.2 Complexity of Fast ComCQ. This section gives the time and space complexity of the Fast ComCQ algorithm proposed in Section 5.5.

Time Complexity: Fast ComCQ employs Fagin's **Threshold Algorithm (TA)** for incremental best-first search. Its worst-case time complexity is $O(N(\log k + T))$ due to a maximum of $O(N)$ iterations. Within each iteration, the process of updating the priority queue incurs a time complexity of $O(\log k)$. Computing the temporal aggregation (step (c)) takes $O(T)$ among T time slices using the hashing property. The evaluation of the terminating condition, i.e., calculating the upper bound score from each time slice, requires $O(T)$ time. Therefore, the overall time complexity of Fast ComCQ is $O(N(logk + T))$.

Space Complexity: Fast ComCQ requires storing the entries from each time slice in the memory during query processing, which takes $O(N)$ storage. Additionally, Fast ComCQ maintains a priority queue for the best-first search, which takes $O(k)$. In reality, $N >> k$. Thus, the time complexity of Fast ComCQ is $O(N)$.

6.2.3 Complexity of Approximate ComCQ. This section gives the time and space complexity of the Fast ComCQ algorithm proposed in Section 5.6. The complexity analysis of Approximate ComCQ is similar to that of Fast ComCQ.

Time Complexity: The time complexity of Approximate ComCQ is $O(\alpha N(\log k + T))$ due to a maximum of $O(\alpha N)$ iterations. The complexity from iteration is $\log k + T$, for the same reason explained in Fast ComCQ.

Space Complexity: Approximate ComCQ takes $O(\alpha N)$ storage to store all the entries from the shortened time slices. Additionally, Approximate ComCQ maintains a priority queue for the best-first search, which takes $O(k)$, which is the same as Fast ComCQ. In reality, $\alpha N >> k$. Thus, the time complexity of Fast ComCQ is $O(\alpha N)$.

Table 3. Parameter Values

Parameter	Settings
Community Size	10k, 100k, 500k, 1M, **2M**
Data Size (millions)	20, 40, 60, **80**
k	10, 50, 100, 500, **1,000**
Time Interval (days)	1, 7, **14**, 28, 56, 84
α	10, 50, 100, 500, **1,000**, 10,000
Keywords	1, **2**, 3, 4, 5, 6
Buffer Size (days)	0, 10, **20**, 40
Δ Users	1k, 10k, 50k, 100k
Spatial Range (kilometer)	**10**, 20 , 40, 80

Table 4. Evaluation Dataset Statistics

Dataset	20M	40M	60M	80M
Interactions	492M	930M	1,428M	1,904M

7 EXPERIMENTAL EVALUATION

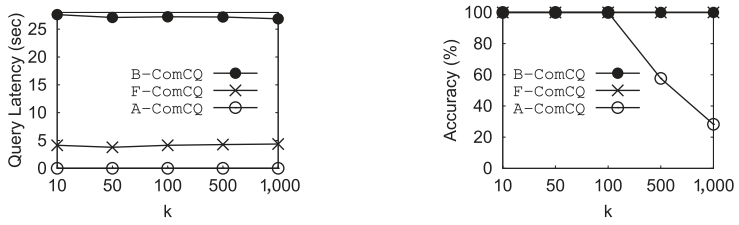
This section presents the experimental evaluation of the indexing framework and the *ComCQ* algorithms as discussed in the previous sections. Section 7.1 explains the experimental settings and the evaluation datasets. Sections 7.2– 7.5 evaluate query latency, indexing scalability, storage overhead, and community dynamism, respectively.

7.1 Experimental Setup

We evaluate the proposed indexing framework and query processing for different performance measures, namely, indexing scalability, storage overhead, query latency, and query accuracy (in case of the approximate algorithm *A-ComCQ*). Table 3 summarizes the evaluation parameters with default values marked as bold. The community size is the number of community users, and data size is the number of posts that the users can interact with. The buffer size is the maximum number of time slices that are buffered in main memory. Δ users are the number of users who join or leave a established community. The rest are the query parameters, except α, which is an *A-ComCQ* query processor parameter. All experiments are based on Java 14 implementation and using an Intel Xeon(R) server with CPU E5-2637 v4 (3.50 GHz) and 128 GB RAM running Ubuntu 16.04. Each index time slice represents 1 day.

Evaluation data. We have collected historical tweets from public Twitter APIs. Then, four datasets, of sizes 20, 40, 60, and 80 million tweets, are composed for our evaluation. These tweets are posted by 5.5M unique users and distributed equally to cover a 12-week period (84 days). Each Tweet is represented with ID, keywords, and the number of interactions based on the number of likes, retweets, replies, and quotes. Table 4 summarizes the number of total interactions in each dataset. A random word from the tweet text is attached as a keyword. A synthetic location for each tweet is generated uniformly as a random point within the bounding rectangle of New York City to simulate a compact community spatial proximity. To simulate the community interactions with the tweets, a portion of the interactions is being randomly distributed to the community users calculated based on its size to total number of users. Thus, bigger communities get a bigger portion of interactions.

Query workloads. We generate the query set based on the time interval size. For example, if the time interval size is 7 days, then we randomly generate all the possible queries which their



(a) Query Latency (b) Accuracy

Fig. 6. Varying k.

time intervals are 7 continuous days from day 1 to day 84. Then, we generate keywords query list randomly chosen from the inverted index in each time slice; these keywords must appear in at least 5,000 tweets to avoid rare keywords that are rarely searched. The query center location is generated as a random point within the bounding rectangle of New York City, similar to the evaluation data, with query spatial range indicated in Table 3.

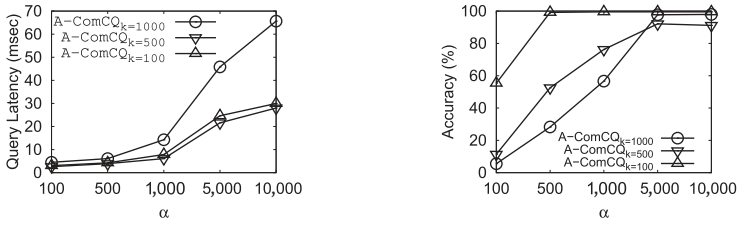
7.2 Query Evaluation

This section evaluates the proposed algorithms to process the *CSTIQ* queries with varying query settings. *B-ComCQ* and *F-ComCQ* algorithms return exact answers, while *A-ComCQ* returns approximate answers. Thus, we report the accuracy of the approximate answers compared to the exact algorithms results. The query latency includes the I/O time to load the data from disk and the query processing in main memory.

Effect of varying k. Figure 6 shows the effect of varying k on *CSTIQ* queries. Figure 6(a) depicts the query latency with varying k for the proposed algorithms. *B-ComCQ* performs the worst among the proposed alternatives, which is 7 times slower than *F-ComCQ* and 2,700 times slower than *A-ComCQ*. *A-ComCQ* provides 385 times faster runtime compared to *F-ComCQ*, while still providing pretty accurate results for practical values of k up to 100. The effect of changing k on query latency is slight, since for a specific time interval, the number of I/O needed is fixed, regardless of k. When k takes a larger value, the query processor takes more to rank the top-k entries. However, the I/O time dominates the query time. Consequently, the query latency is less likely affected by k. Figure 6(b) shows the accuracy of *A-ComCQ* compared to the ground truth of *B-ComCQ* and *F-ComCQ*. Indeed, *A-ComCQ* is very accurate for k up to 100, but not that accurate for larger k values. Given that the default α value is 1,000, this is interpreted that for large k value, the bottom item in the top-k might not be retrieved by *A-ComCQ* within the first 1,000 items of the time slice. This low accuracy significantly improves with changing the value of α to be a multiple of the k value as discussed below.

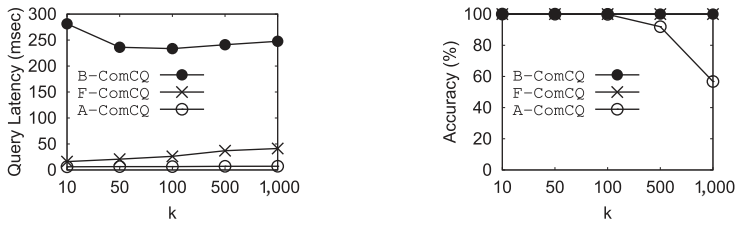
Effect of varying α. The number of processed items in each time slice, α, affects the performance of *A-ComCQ* and trades off query latency with query accuracy. Figure 7 shows the effect of α on both query latency and accuracy for k values 100, 500, and 1,000. Figure 7(a) shows that the query latency of *A-ComCQ* is increasing linearly when the α value is increased. Figure 7(b) depicts the accuracy with varying α values. k and α are correlated. When α value is significantly less than k, for example α=10 and k=1,000, the accuracy score is expected to be worse with low latency, while increasing α improves accuracy while increasing latency. The best tradeoff is when α value is set as three times k, which provides high accuracy with reasonable latency. For practical values of k, up to 100, all experiments show efficient query latency with high accuracy.

Effect of varying k **with keywords**. Figure 8 shows the effect of varying k on *CSTIQ* queries with keywords. Figure 8(a) shows the latency of the three algorithms. The query latency of both



(a) Query Latency (b) Accuracy

Fig. 7. Varying α.



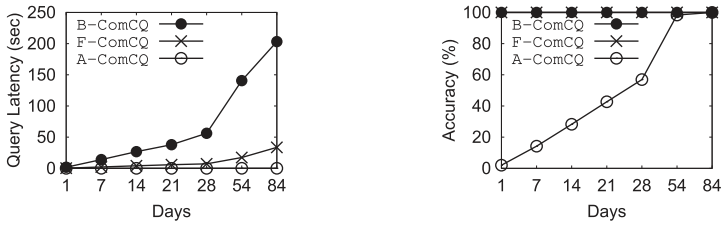
(a) Query Latency (b) Accuracy

Fig. 8. Varying k with keywords.

F-ComCQ and *A-ComCQ* is increasing linearly with increasing k. However, the query latency is significantly less for the same queries without keywords, as shown in Figure 6, because of the power of keyword pruning that is equipped in the community index. Therefore, the query processor utilizes the inverted index to reduce the search space to only tweets that contain the query keywords. Figure 8(b) shows the accuracy scores for *A-ComQ*. *A-ComCQ* is accurate for k values up to 100. The accuracy reduces when default value of α is less than three times k, as discussed before. However, the reduction in accuracy when querying keywords is noticeably less than the reduction in Figure 6(b).

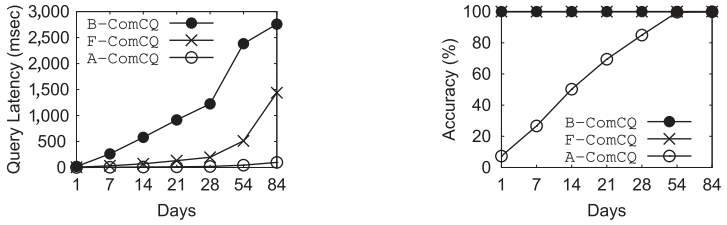
Effect of varying time interval length. Figure 9 shows the performance of the three algorithms with different time interval lengths measured by days. Each day represents a single time slice. Figure 9(a) depicts the query latency. All algorithms encounter longer query latency when the query time interval increases. This is because a longer time interval incurs more I/Os and more number of entries to process in the ranking of the query. Relatively, *F-ComCQ* performs 5–8 times faster, while *A-ComCQ* performs 152–2,693 times faster than *B-ComCQ*. Thus, *A-ComCQ* performs steady with the lowest query latency, and it is up to 400 times faster than *F-ComCQ*, with better scalability over large time periods. Figure 9(b) shows the accuracy of *A-ComQ*. *A-ComCQ* has increasing accuracy with increasing the time interval length, with the default values of k and α. This is because a popular item tends to appear frequently in a continuous time period. An item from the top-k that is not fetched by *A-ComCQ* in one time slice is probably fetched by in another time slice. *A-ComCQ* is primarily designed for long time periods, as the time slices are many and it is an overhead to load the whole lists in memory. So, the accuracy performance of *A-ComCQ* is compliant with this design goal and it works effectively on long periods. From another perspective, querying small- and medium-length time periods calls for increasing the value of α to provide acceptable accuracy, which does not cause a problem, as the number of time slices is small for these queries.

Effect of varying time interval length with keywords. Figure 10 shows the effect of time interval length on *CSTIQ* queries with keywords. Clearly, the query latency is much less than the



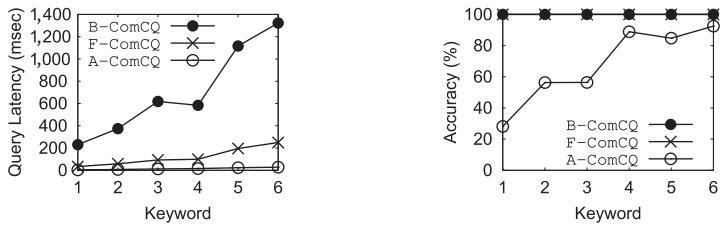
(a) Query Latency (b) Accuracy

Fig. 9. Varying time intervals.



(a) Query Latency (b) Accuracy

Fig. 10. Varying time intervals with keywords.



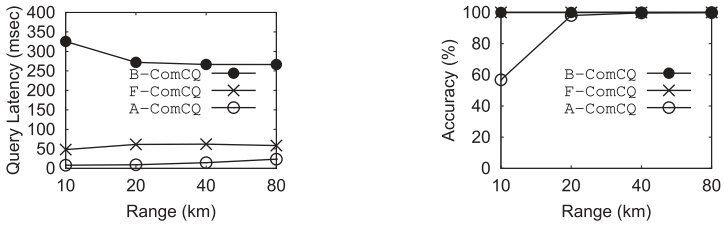
(a) Query Latency (b) Accuracy

Fig. 11. Varying query range.

same query without keywords, as shown in Figure 9; the reason is utilizing the equipped inverted index, which filters out the objects based on the existence of the query keywords. *A-ComCQ* still achieves the lowest query latency with all different time intervals with similar latency versus accuracy tradeoff to Figure 9.

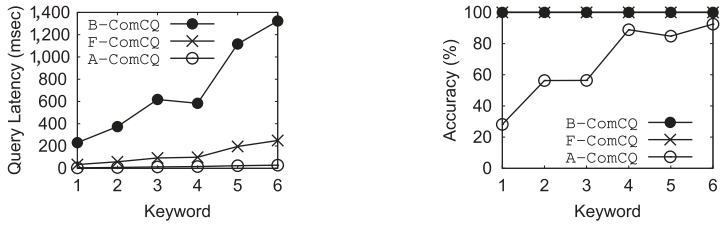
Effect of varying spatial range. Figures 11 and 12 show the performance of the three algorithms versus different query spatial range with and without keywords, respectively. All algorithms show steady query latency as the query spatial range varies; this is because changing the spatial range does not affect the number of I/Os, which is the dominating factor in the query processing. Queries with keywords have much less latency due to the inverted index pruning. *B-ComCQ* performs worse than the other alternatives. Figures 11(b) and 12(b) show that *A-ComCQ* is more accurate for large spatial ranges; this is because under small spatial range, the top items that *A-ComCQ* retrieves from each time slice might not have that many items that satisfy the query range. This calls for increasing α value for small spatial ranges.

Effect of varying keywords. Figure 13 shows the performance of the three algorithms under different number of keywords. The query latency of all algorithms increases as the number of keywords increases; this is because when the number of keyword increases, we access more on



(a) Query Latency (b) Accuracy

Fig. 12. Varying query range with keywords.



(a) Query Latency (b) Accuracy

Fig. 13. Varying keywords.

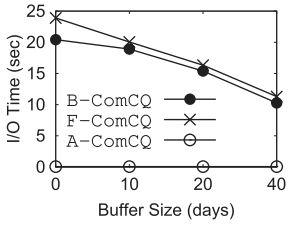


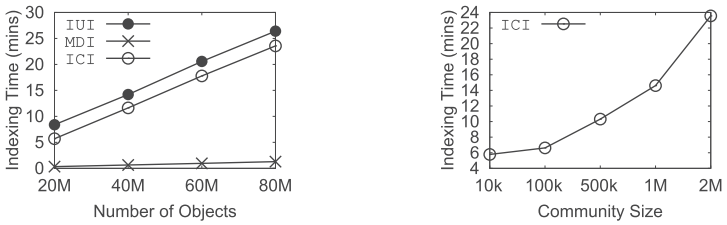
Fig. 14. Varying Buffer Sizes.

the inverted index, which takes more time. Figure 13(b) shows that *A-ComCQ* is more accurate with large number of keywords; this is because under multiple keywords, the marginal error brought by *A-ComCQ* is minimized.

Effect of varying buffer sizes. Figure 14 shows the effect of the size of the buffer on the *CSTIQ* queries performance in terms of I/O time. Clearly, *B-ComCQ* and *F-ComCQ* algorithms take the most advantage of the buffer and avoid retrieving the data from the disk if the data exists in the buffer due to their high latency compared to *A-ComCQ*. If the size of the buffer increases, then the I/O time decreases significantly.

7.3 Indexing Scalability

This section evaluates the scalability of the proposed indexes in terms of time required to build the index. We evaluate the three main indexing components: the community index for **indexing community interactions (*ICI*)**, the user index for **indexing user interactions (*IUI*)**, and the **master data index (*MDI*)**. Figure 15(a) shows the index scalability with different number of objects. All indexes require more time to build the index when the number of objects increases. However, *IUI* and *ICI* need additional processing for the interactions while *MDI* only processes



(a) Varying Dataset (b) Varying Community Sizes

Fig. 15. Indexing scalability.

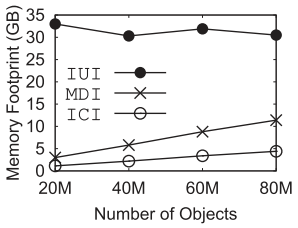


Fig. 16. Storage overhead with varying datasets.

the objects and indexes them in batching. The indexing time includes the I/O time to store the expired time slices to the disk for *IUI* and *ICI* in addition to the I/O time of *MDI* to store the objects when the buffer reaches its capacity. Figure 15(b) shows the impact of the community size on the indexing time. When the number of users is increased, the indexing time is increasing due to the increasing number of interactions per user. Thus, when community size is 10k, the number of interaction is tremendously less than when the community size is 1M.

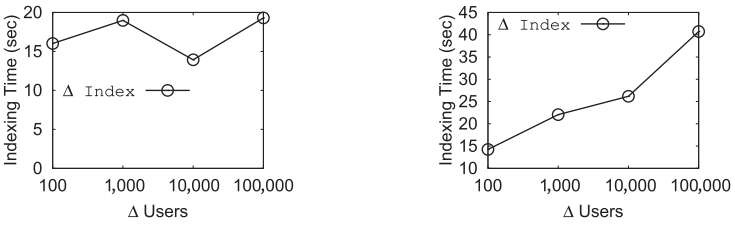
7.4 Storage Overhead

This section evaluates the storage overhead for the proposed indexes. Figure 16 shows the storage overhead in GB for the indexes *IUI*, *ICI*, and *MDI*. The interaction indexes (*IUI* and *ICI*) are stable even when the number of objects is increased. The interaction indexes depend on the number of users and the number of interactions; thus, the overall storage overhead is not affected by the number of objects. However, *MDI* is increasing linearly when the number of objects is increased, as the number of objects heavily affects *MDI*.

7.5 Community Dynamism

In this section, we evaluate the overhead of community dynamism where users join and leave the community. U_Δ represents the list of users with changing community membership.

Effect of indexing U_Δ users. Figure 17 shows the indexing time with varying U_Δ users' sizes. Figure 17(a) shows the effect of the U_Δ users in case the *CSTIQ* query does not include any keywords. The indexing time for small U_Δ users is dominant by the I/O time, as we need to retrieve all the time slices of *IUI* from the disk to build the index for the new U_Δ users. However, when the U_Δ users is significantly bigger, the CPU time is increased accordingly. Figure 17(b) depicts the indexing time for the U_Δ users when *CSTIQ* query includes keywords. The effect of keywords is minimal when the size of U_Δ users is relatively small. However, the effect is very obvious when the size of U_Δ users is very large. For example, indexing time for U_Δ users size equal 100k is double the indexing time without keywords. The additional overhead comes from the I/O of retrieving the data from disk utilizing the *MDI* index.



(a) Varying Δ (b) Varying Δ with keywords

Fig. 17. Indexing Δ users.

8 CONCLUSION

This article has introduced community-centric query, which returns top-k objects that a specific community interacted with the most given a time interval, an optional spatial range, and a set of keywords. We proposed a novel indexing framework that provides efficient resource management and scalable query processing. Moreover, we propose three algorithms, exact and approximate, that efficiently process the queries by utilizing the underling indexing framework. Our techniques support community dynamism, which allows users to join and leave communities over time. We avoid rebuilding the community index when users change their community memberships. We evaluated the proposed techniques on a real Twitter dataset and have shown their efficiency to handle large communities.

REFERENCES

[1] [n. d.]. Retrieved from http://neo4j.org/
[2] [n. d.]. Facebook. Retrieved from https://www.facebook.com/
[3] [n. d.]. LinkedIn. Retrieved from https://www.linkedin.com/
[4] 2019. List of Virtual Communities with More Than 1 Million Users. Retrieved from http://www.worldheritage.org/articles
[5] Nitin Agarwal, Huan Liu, Lei Tang, and Philip S. Yu. 2008. Identifying the influential bloggers in a community. In *Proceedings of the ACM International Conference on Web Search and Data Mining*. 207–218.
[6] Nesreen K. Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. 2014. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the SIGKDD Conference*. 1446–1455.
[7] Christopher Aicher, Abigail Z. Jacobs, and Aaron Clauset. 2015. Learning latent block structure in weighted networks. *J. Complex Netw.* 3, 2 (2015), 221–248.
[8] Abdullah Almaatouq, Ahmad Alabdulkareem, Mariam Nouh, Erez Shmueli, Mansour Alsaleh, Vivek K. Singh, Abdulrahman Alarifi, Anas Alfaris, and Alex Pentland. 2014. Twitter: Who gets caught? Observed trends in social micro-blogging spam. In *Proceedings of the ACM Conference on Web Science*. 33–41.
[9] Abdulaziz Almaslukh, Yongyi Liu, and Amr Magdy. 2021. Scalable spatio-temporal top-k community interactions query. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*. 293–296.
[10] Abdulaziz Almaslukh and Amr Magdy. 2019. Temporal geo-social personalized search over streaming data. In *Proceedings of the SIGSPATIAL Conference*. 189–198.
[11] Nicky Antonius and L. Rich. 2013. Discovering collection and analysis techniques for social media to improve public safety. *Int. Technol. Manag. Rev.* 3, 1 (2013), 42–53.
[12] P. Bahl, R. Chancre, and J. Dungeon. 2016. GraphJet: Real-time content recommendations at Twitter. In *VLDB Endowment*, VLDB Endowment, 1281–1292.
[13] Radoslav Baltezarevic, Borivoje Baltezarevic, Piotr Kwiatek, and Vesna Baltezarevic. 2019. The impact of virtual communities on cultural identity. *Symposion* 6, 1 (2019), 7–22.
[14] Hamid Ahmadi Beni and Asgarali Bouyer. 2020. TI-SC: Top-k influential nodes selection based on community detection and scoring criteria in social networks. *J. Amb. Intell. Hum. Comput.* 11 (2020), 1–20.
[15] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. 2013. Spatial keyword query processing: An experimental evaluation. *Proc. VLDB Endow.* 6, 3 (2013), 217–228.
[16] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. 2018. Maximum co-located community search in large scale social networks. *Proc. VLDB Endow.* 11, 10 (2018), 1233–1246.

[17] Mingming Chen, Konstantin Kuzmin, and Boleslaw K. Szymanski. 2014. Community detection via maximization of modularity and its variants. *IEEE Trans. Computat. Soc. Syst.* 1, 1 (2014), 46–65.
[18] Chennai Floods 2017. How Twitter, Facebook, WhatsApp and Other Social Networks Are Saving Lives During Disasters. Retrieved from http://www.huffingtonpost.in/2017/01/31/how-twitter-facebook-whatsapp-and-other-social-networks-are-sa_a_21703026/
[19] Coronavirus 2020. Here's How Social Media Can Combat the Coronavirus Infodemic. Retrieved from https://www.technologyreview.com/s/615368/facebook-twitter-social-media-infodemic-misinformation/
[20] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online search of overlapping communities. In *Proceedings of the SIGMOD Conference*. 277–288.
[21] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *Proceedings of the SIGMOD Conference*. 991–1002.
[22] Thu-Lan Dam, Sean Chester, Kjetil Nørvåg, and Quang-Huy Duong. 2021. Efficient top-k recently-frequent term querying over spatio-temporal textual streams. *Inf. Syst.* 97 (2021), 101687.
[23] Daniel J. DiTursi, Gaurav Ghosh, and Petko Bogdanov. 2017. Local community detection in dynamic networks. In *Proceedings of the International Conference on Data Mining*. 847–852.
[24] Jian Dong, Bin Chen, Chuan Ai, Liang Liu, and Fang Zhang. 2018. A degree-based distributed label propagation algorithm for community detection in networks. In *Proceedings of the 4th ACM SIGSPATIAL International Workshop on Safety and Resilience*. 1–5.
[25] Facebook Stats 2019. 53 Incredible Facebook Statistics and Facts. Retrieved from https://www.brandwatch.com/blog/facebook-statistics/
[26] Facebook Stats Teen 2020. Distribution of Facebook Users in the United States. Retrieved from https://www.statista.com/statistics/187549/facebook-distribution-of-users-age-group-usa/
[27] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.* 66, 4 (2003), 614–656.
[28] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective community search over large spatial graphs. *Proc. VLDB Endow.* 10, 6 (2017), 709–720.
[29] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *Proc. VLDB Endow.* 9, 12 (2016), 1233–1244.
[30] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *VLDB J.* 29, 1 (2020), 353–392.
[31] Greg Fisher. 2019. Online communities and firm advantages. *Acad. Manag. Rev.* 44, 2 (2019), 279–298.
[32] Nick Fox and Chris Roberts. 1999. GPs in cyberspace: The sociology of a "virtual community." *Sociolog Rev* 47, 4 (1999), 643–671.
[33] Theodore Georgiou, Amr El Abbadi, and Xifeng Yan. 2017. Extracting topics with focused communities for social content recommendation. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work and Social Computing*. 1432–1443.
[34] Steve Harenberg, Gonzalo Bello, La Gjeltema, Stephen Ranshous, Jitendra Harlalka, Ramona Seay, Kanchana Padmanabhan, and Nagiza Samatova. 2014. Community detection in large-scale networks: A survey and empirical evaluation. *Wiley Interdiscip. Rev.: Computat. Stat.* 6, 6 (2014), 426–439.
[35] Abdeltawab M. Hendawi and Mohamed F. Mokbel. 2012. Predictive spatio-temporal queries: A comprehensive survey and future directions. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems*. 97–104.
[36] Tuan-Anh Hoang-Vu, Huy T. Vo, and Juliana Freire. 2016. A unified index for spatio-temporal keyword queries. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 135–144.
[37] Han Hu, NhatHai Phan, Xinyue Ye, Ruoming Jin, Kele Ding, Dejing Dou, and Huy T. Vo. 2019. DrugTracker: A community-focused drug abuse monitoring and supporting system using social media and geospatial data (demo paper). In *Proceedings of the SIGSPATIAL Conference*. 564–567.
[38] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the SIGMOD Conference*. 1311–1322.
[39] Hurricane Dorian 2019. How Social Media Is Helping Survivors of Hurricane Dorian in the Bahamas. Retrieved from https://www.newyorker.com/news/news-desk/how-social-media-is-helping-survivors-of-hurricane-dorian-in-the-bahamas
[40] IG Stats Teen 2021. Instagram by the Numbers: Stats, Demographics & Fun Facts. Retrieved from https://www.omnicoreagency.com/instagram-statistics/
[41] IG Stats Teen 2021. Most Popular Social Networks of Teenagers in the United States from Fall 2012 to Fall 2020. Retrieved from https://www.statista.com/statistics/250172/social-network-usage-of-us-teens-and-young-adults/

[42] Martin Junghanns, André Petermann, Martin Neumann, and Erhard Rahm. 2017. Management and analysis of big graph data: Current systems and open challenges. In *Handbook of Big Data Technologies*, Springer, 457–505.
[43] Bisma S. Khan and Muaz A. Niazi. 2017. Network community detection: A review and visual survey. *arXiv preprint arXiv:1708.00977* (2017).
[44] Jungeun Kim and Jae-Gil Lee. 2015. Community detection in multi-layer graphs: A survey. *ACM SIGMOD Rec.* 44, 3 (2015), 37–48.
[45] Reinout Kleinhans, Maarten Van Ham, and Jennifer Evans-Cowley. 2015. Using social media and mobile technologies to foster engagement and self-organization in participatory urban planning and neighbourhood governance. *Planning Practice & Research* (2015).
[46] Jimmy Lin and Michael Schatz. 2010. Design patterns for efficient graph algorithms in MapReduce. In *Proceedings of the 8th Workshop on Mining and Learning with Graphs*.
[47] Xiping Liu, Changxuan Wan, Neal N. Xiong, Dexi Liu, Guoqiong Liao, and Song Deng. 2018. What happened then and there: Top-k spatio-temporal keyword query. *Inf. Sci.* 453 (2018), 281–301.
[48] Zhi Liu and Yan Huang. 2014. Community detection from location-tagged networks. In *Proceedings of the SIGSPATIAL Conference*. 525–528.
[49] Ahmed R. Mahmood, Sri Punni, and Walid G. Aref. 2019. Spatio-temporal access methods: A survey (2010–2017). *GeoInformatica* 23 (2019), 1–36.
[50] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the SIGMOD Conference*. 135–146.
[51] Deepanshu Malhotra. 2021. Community detection in complex networks using link strength-based hybrid genetic algorithm. *SN Comput. Sci.* 2, 1 (2021), 1–16.
[52] Shohei Matsugu, Hiroaki Shiokawa, and Hiroyuki Kitagawa. 2021. Fast algorithm for attributed community search. *Journal of Information Processing* 29 (2021), 188–196.
[53] Daniel Memmi. 2008. The nature of virtual communities. In *Cognition, Communication and Interaction*, Springer, 70–82.
[54] Robert Munro. 2013. Crowdsourcing and the crisis-affected community. *Inf. Retriev.* 16, 2 (2013), 210–266.
[55] Seth A. Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. 2014. Information network or social network? The structure of the Twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*. 493–498.
[56] Preslav Nakov, Doris Hoogeveen, Lluís Màrquez, Alessandro Moschitti, Hamdy Mubarak, Timothy Baldwin, and Karin Verspoor. 2019. SemEval-2017 task 3: Community question answering. *arXiv preprint arXiv:1912.00730* (2019).
[57] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. 2012. Community detection in social media. *Data Min. Knowl. Discov.* 24, 3 (2012), 515–554.
[58] Michel Plantié and Michel Crampes. 2013. Survey on social community detection. In *Social Media Retrieval*, Springer, 65–85.
[59] Constance Elise Porter. 2004. A typology of virtual communities: A multi-disciplinary foundation for future research. *J. Comput.-mediat. Commun.* 10, 1 (2004), JCMC1011.
[60] Jenny Preece and Diane Maloney-Krichmar. 2005. Online communities: Design, theory, and practice. *J. Comput.-mediat. Commun.* 10, 4 (2005), JCMC10410.
[61] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. 2014. Scalable big graph processing in MapReduce. In *Proceedings of the SIGMOD Conference*. 827–838.
[62] Anjum Razzaque and Tillal Eldabi. 2018. Assessing the impact of physicians' virtual communities on their medical decision making quality. In *Proceedings of the 51st Hawaii International Conference on System Sciences*.
[63] Joao B. Rocha-Junior, Orestis Gkorgkas, Simon Jonassen, and Kjetil Nørvåg. 2011. Efficient processing of top-k spatial keyword queries. In *Proceedings of the 12th International Symposium on Advances in Spatial and Temporal Databases (SSTD'11)*. Springer, 205–222.
[64] Arif Mohaimin Sadri, Samiul Hasan, Satish V. Ukkusuri, and Manuel Cebrian. 2017. Understanding information spreading in social media during Hurricane Sandy: User activity and network properties. *arXiv preprint arXiv:1706.03019* (2017).
[65] Arif Mohaimin Sadri, Samiul Hasan, Satish V. Ukkusuri, and Manuel Cebrian. 2020. Exploring network properties of social media interactions and activities during Hurricane Sandy. *Transport. Res. Interdiscip. Perspect.* 6 (2020), 100143.
[66] Arif Mohaimin Sadri, Samiul Hasan, Satish V. Ukkusuri, and Juan Esteban Suarez Lopez. 2018. Analysis of social interaction network properties and growth on Twitter. *Soc. Netw. Anal. Min.* 8 (2018), 1–13.
[67] Bin Shao, Haixun Wang, and Yatao Li. 2012. The trinity graph engine. *Microsoft Res.* 54 (2012), 4.
[68] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the SIGKDD Conference*. 939–948.

[69] Ivan Srba and Maria Bielikova. 2016. A comprehensive survey and classification of approaches for community question answering. *ACM Trans. Web* 10, 3 (2016), 1–63.
[70] Beiming Sun and Vincent T. Y. Ng. 2012. Identifying influential users by their postings in social networks. In *Ubiquitous Social Media Analysis*. 128–151.
[71] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGraph: An efficient relational-based property graph store. In *Proceedings of the SIGMOD Conference*. 1887–1901.
[72] Lei Tang and Huan Liu. 2010. Community detection and mining in social media. *Synthes. Lect. Data Min. Knowl. Discov.* 2, 1 (2010), 1–137.
[73] Xuning Tang and Christopher C. Yang. 2010. Identifying influential users in an online healthcare social network. In *Proceedings of the IEEE International Conference on Intelligence and Security Informatics*. 43–48.
[74] G. Alan Wang, Jian Jiao, Alan S. Abrahams, Weiguo Fan, and Zhongju Zhang. 2013. ExpertRank: A topic-aware expert finding algorithm for online knowledge communities. *Decis. Supp. Syst.* 54, 3 (2013), 1442–1451.
[75] Hongzhi Wang, Zhixin Qi, Lei Zheng, Yun Feng, Junfei Ouyang, Haoqi Zhang, Xiangxi Zhang, Ziming Shen, and Shirong Liu. 2020. April: An automatic graph data management system based on reinforcement learning. In *Proceedings of the ACM International on Conference on Information and Knowledge Management*. 3465–3468.
[76] Michael J. Welch, Uri Schonfeld, Dan He, and Junghoo Cho. 2011. Topical semantics of Twitter links. In *Proceedings of the 4th ACM International Conference on Web Search and Data Mining*. 327–336.
[77] Joyce Jiyoung Whang, David F. Gleich, and Inderjit S. Dhillon. 2013. Overlapping community detection using seed set expansion. In *Proceedings of the ACM International on Conference on Information and Knowledge Management*. 2099–2108.
[78] Kai Zheng, Han Su, Bolong Zheng, Shuo Shang, Jiajie Xu, Jiajun Liu, and Xiaofang Zhou. 2015. Interactive top-k spatial keyword queries. In *Proceedings of the IEEE 31st International Conference on Data Engineering*. IEEE, 423–434.

Received 1 March 2023; revised 11 December 2023; accepted 1 February 2024