Scalable Spatio-Temporal Top-k Community Interactions Query

Abdulaziz Almaslukh*
King Saud University
Riyadh, Saudi Arabia
aalmaslukh@ksu.edu.sa

Yongyi Liu
University of California, Riverside
Riverside, California
yliu786@ucr.edu

Amr Magdy**
University of California, Riverside
Riverside, California
amr@cs.ucr.edu

ABSTRACT

The excessive amount of data that online users produce through social media platforms provides valuable insights about users and communities at scale. 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 paper introduces a new analytical query that reveals the top-k posts of interest of a given user community over a period of time and in a certain location. We propose a novel indexing framework that captures the interactions of community users to provide a low query latency. Moreover, we propose efficient query algorithms that 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*; **Data management systems**.

KEYWORDS

community, query processing, spatio-temporal query

ACM Reference Format:
Abdulaziz Almaslukh, Yongyi Liu, and Amr Magdy. 2021. Scalable Spatio-Temporal Top-k Community Interactions Query. In *29th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '21), November 2–5, 2021, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3474717.3483962

1 INTRODUCTION

Online communities have become more popular with the advancement of user-generated data platforms. They are rich with useful and important data. The research community has paid significant attention to detecting [7] and searching communities [5] that are homogeneous and have shared characteristics in common. Nevertheless, analyzing a single community has got very little attention. For example, a social scientist might analyze the US teenagers on Facebook to find out what they interacted with during the last week on different topics such as bullying, youth suicide, and COVID-19 pandemic. This kind of analysis is crucial in understanding, making the right decisions, and offering better services to the target community. In fact, there are tens of online communities with multi-million users [1], each of them has lots of large sub-communities, and it is recognized that analyzing these communities is of high importance in various applications [2, 6]. However, existing querying techniques are not scalable to perform such analysis on large online communities that involve millions of users.

In this work, we propose a spatio-temporal community query that finds the top-k posts with which the given community has interacted during a given time and within a given spatial range. The 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, the main challenge is that the community users' interactions with the posts are huge in number and constantly increasing. For instance, Facebook users upload on average 240K photos/min and generate 4M likes/min [3]. Therefore, digesting this information is a major challenge to reduce the query latency and minimize the system resources overhead.

To address the challenge, we devise novel scalable indexing and query processing that deal with communities as whole units instead of processing individual users' data. 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 online communities that returns the top-k posts a community has interacted with.
- We propose a novel indexing paradigm with multiple components to efficiently support established communities.
- We develop different query processing techniques to efficiently process the query.
- We provide an extensive experimental evaluation on real data to show the superiority of our proposed techniques.

The rest of this paper is organized as follows. Section 2 presents the problem definition. Sections 3 and 4 detail the proposed community indexing and query processing techniques, respectively. Section 5 experimentally evaluate our techniques. Finally, Section 6 concludes the paper.

*The work has been performed while the first author is 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.

SIGSPATIAL '21, November 2–5, 2021, Beijing, China

ACM ISBN 978-1-4503-8664-7/21/11…$15.00
https://doi.org/10.1145/3474717.3483962

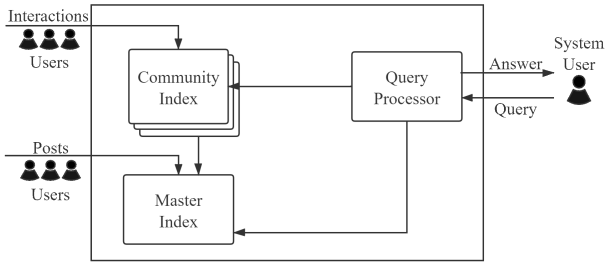


Figure 1: CSTIQ Framework

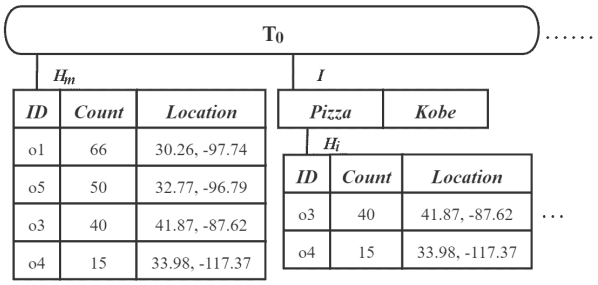


Figure 2: Community Index Structure

2 PROBLEM DEFINITION

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. In this work, we use the terms post and object interchangeably. A *virtual community (C)* is defined as a set of users $\{u_1, u_2, u_3, ..., u_n\}$ 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. 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 that 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 community 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 period. Both keywords and locations are optional, which enables the query to be flexible in different aspects.

3 CSTIQ INDEXING

In this section, we introduce *Community Spatio-Temporal Indexing Query (CSTIQ)* framework to serve our query as defined in Section 2. Figure 1 shows an overview of *CSTIQ* framework. It consists of two 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 has efficient hash insertion $O(1)$ and temporally sliced to handle an excessive volume of interactions. The *community index* stores only post ids and locations. These ids are used to retrieve data from a second indexing component, called the *master index*, that stores all available information about posts, e.g., user profile or keywords.

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, ..., 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., 24hrs, 12hrs, or 1hr, 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. On the other hand, for the query that specifies keyword parameter, I will be used to process the query efficiently.

Figure 2 shows the structure of an example community index. The figure shows one time slice labeled as T_0. 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 in community C interacts with an object oid, C's corresponding index is updated accordingly. The in-memory index updates two data structures: the master hash index H_m and the inverted index I. First, oid is looked up in H_m. If oid is found, 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 every associated keyword, it is

added to I if not existing. Then, its H_i hash structure is updated in a similar way to H_m.

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. 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.

4 QUERY PROCESSING

This section describes the query processing of *CSTIQ* query defined in Section 2 utilizing the indexing framework introduced in Section 3. We propose two algorithms to process *CSTIQ* query namely *baseline ComCQ* and *fast ComCQ*. The rest of this section introduces a high-level query processing framework that is utilized for both algorithms, and then we detail the specifics for both.

4.1 Query Processing Framework

This section introduces a two-step query processing framework. The first step is shared among both *baseline ComCQ* and *fast ComCQ*, while the second step differs based on the algorithm. Every *CSTIQ* query takes a community C parameter as an input. We retrieve the Community C index and feed it to the following steps:

(1) Step 1: Temporal and keyword filtering. Given the community C index, 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, the query processor retrieves H_i hash structures that are corresponding to every query keywords and merge 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, 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. Finally, the lists L_j are fed to Step 2.

(2) Step 2: Spatio-temporal aggregation. Given the lists of posts that are retrieved in Step 1, these lists are processed to return the top-k posts that the community C interacted with during the query time interval and within the query spatial range. We have two different variations of our query processing technique where the first one is the naive processing technique and the second is the optimized processing technique. The following sections detail both.

4.2 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 1. *B-ComCQ* creates a new hash structure 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, 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.

4.3 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 [4] called *fast ComCQ* (*F-ComCQ*). *F-ComCQ* does not need to access every entry in every list L_j. Instead, it smartly prunes entries that surly 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 1, *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 equals 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 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, 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.

(d) **Top-k answer update**. If size of *Ans* list < k, insert e_Q into the ordered list *Ans*. Once the size of *Ans* grows to k and larger, we maintain *lower_bound* as the k^{th}, i.e., lowest, score in *Ans*. If total interactions of e_Q > *lower_bound*, we remove the k^{th} 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 $\geq k$ and *lower_bound* $\geq$ *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.

5 EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of the indexing framework and the *ComCQ* algorithms as discussed in the previous sections. Section 5.1 explains the experimental settings and the evaluation datasets. Sections 5.2 evaluates query latency.

5.1 Experimental Setup

We evaluate the proposed indexing framework and query processing for query latency. Table 1 summarizes the evaluation parameters with default values marked as bold. All experiments are based on Java 14 implementation and using an Intel Xeon(R) server with CPU E5-2637 v4 (3.50 GHz) and 128GB RAM running Ubuntu 16.04. Each index time slice represents 1 day.

Parameter	Settings
k	10, 50, 100, 500, **1000**
Time Interval (days)	1, 7, **14**, 28, 56, 84

Table 1: Parameters Values

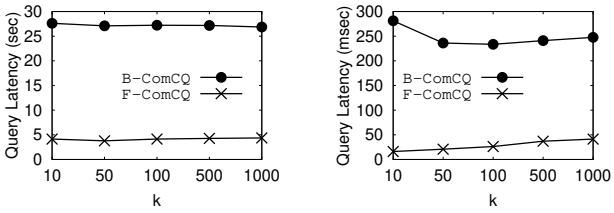
Evaluation data. We have collected historical tweets from public Twitter APIs of size 80 million tweets. These tweets are posted by 5.5M unique users and distributed equally to cover 12 weeks (84 days) period. Each Tweet is represented with id, keywords, and the number of interactions based on the number of likes, retweets, replies, and quotes. The total number of interactions in this dataset is 1904M. 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. In order 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 the total number of users. In this experiment, we set the community size to be 2M, the spatial range to be 10km, and the number of keywords to be 2.

Query workloads. We generate the query set based on the time interval size. For example, if the time interval size is 7 days, we randomly generate all the possible queries with time intervals having 7 continuous days. Then, we generate keywords query list randomly chosen from the inverted index in each time slice, these keywords must appear in at least 5000 tweets to avoid rare keywords that are rarely searched. The query center is generated as a random point within the bounding rectangle of New York City.

5.2 Query Evaluation

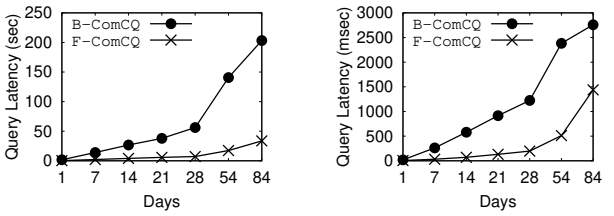
This section evaluates the proposed algorithms to process *CSTIQ* queries. The query latency includes the I/O time to load the data from disk and the query processing in main memory.

Effect of varying k. Figure 3 shows the effect of varying k on *CSTIQ* queries. Figure 3a shows the latency when varying k on *CSTIQ* queries with no keyword. *F-ComCQ* is 7 times faster than *B-ComCQ* due to the efficient pruning. 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 time to rank the top-k entries. However, the I/O time dominates the query time. Consequently, the query latency is only slightly affected by k. Figure 3b shows the latency of the two algorithm when varying k on *CSTIQ* queries



(a) Latency - no Keywords **(b) Latency - Keywords**

Figure 3: Varying k



(a) Latency - no Keyword **(b) Latency - Keyword**

Figure 4: Varying time intervals

with keywords. The query latency is significantly less for the same queries without keywords as shown in Figure 3a because of the effcient utilization of the inverted index.

Effect of varying time interval length. Figure 4 shows the effect of varying rime interval length on *CSTIQ* queries. Figure 4a depicts the query latency with no keyword. Both algorithms encounter longer query latency when the query time interval increases. This is because a longer time interval incurs more I/Os and more entries to process in the ranking of the query. *F-ComCQ* performs 5-8 times faster than *B-ComCQ*. Thus, *F-ComCQ* demonstrates better scalability over large time periods. Figure 4b shows the latency on *CSTIQ* queries with keywords. Clearly, the query latency is much less than the same query without keywords as shown in Figure 4a. The reason is efficiently utilizing the equipped inverted index to reduce the search space.

6 CONCLUSION

This paper has introduced community-centric query that returns the top-k objects that a specific community interacted the most given a time interval, an optional spatial range and a set of keywords. We proposed a novel indexing framework and query algorithms that efficiently process the community queries. We evaluated the proposed techniques on real Twitter dataset and have shown their efficiency to handle large communities.

REFERENCES

[1] List of Virtual Communities with More Than 1 Million Users. http://www.worldheritage.org/articles/eng/List_of_virtual_communities_with_more_than_1_million_users, 2019.
[2] R. Baltezarevic, B. Baltezarevic, P. Kwiatek, and V. Baltezarevic. The Impact of Virtual Communities on Cultural Identity. *Symposion*, 6(1):7–22, 2019.
[3] 53 Incredible Facebook Statistics and Facts. https://www.brandwatch.com/blog/facebook-statistics/, 2019.
[4] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.
[5] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A Survey of Community Search Over Big Graphs. *The VLDB Journal*, 29(1):353–392, 2020.
[6] G. Fisher. Online Communities and Firm Advantages. *Academy of Management Review*, 44(2):279–298, 2019.
[7] B. S. Khan and M. A. Niazi. Network Community Detection: A Review and Visual Survey. *arXiv preprint arXiv:1708.00977*, 2017.