Efficient Top-k Recently-Frequent Term Querying over Spatio-Temporal Textual Streams

Thu-Lan Dam[a], Sean Chester[b], Kjetil Nørvåg[a], Quang-Huy Duong[a]

[a]*Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway*
[b]*Department of Computer Science, University of Victoria, Victoria, Canada*

Abstract

Massive amounts of data with spatio-temporal-textual information are being generated due to the proliferation of GPS-equipped mobile devices. Much of this data are social media posts, often used to share and spread personal updates and news. Exploring valuable information from a dynamic collection of social posts is of great interest and has attracted many studies. However, because the size of data is huge, the existing methods mostly work with the time window model where the old data is discarded. In this work, we introduce the task of efficiently discovering the top-k most popular terms within a user specified bounded region over a stream of social posts, where the recent posts are more important than the old ones. To make this feasible, we propose a hybrid index structure and algorithms to efficiently answer such top-k queries. Our index employs a spatial index augmented by top-k time-weighted term lists and a bulk updating technique to support fast digestion of social post streams. Further, these top-k term lists are employed in the aggregation step to produce the final results so that incoming queries can be efficiently processed. An extensive experimental study with a large collection of social posts shows that the proposed methods are capable of both online aggregation and accurate query processing.

Keywords: Frequent terms, Time-weighted, Spatio-temporal query, Top-k query, Spatial index, Spatio-Temporal Textual Stream.

1. Introduction

With the rapid proliferation of both GPS-enabled mobile devices and online social media such as Twitter, Facebook and Sina Weibo, huge amounts of textual data are being generated with both spatial coordinates and timestamps. Such data, referred to as *social posts*, enables exciting real-time spatio-temporal-textual analytics such as detecting trending terms in an area [1, 2, 3, 4, 5]. These terms, which could include new hash tags and latent topics, can provide data scientists and journalists with early leads on current and emerging events.

Nowadays, more and more people use social networks such as Twitter to convey thoughts and interpret daily events taking place in the surrounding space. The regular and highlight events will get attention and have interest from many people. Social network users frequently post and share messages about the events, and the users discuss the events on social media, where the featured keywords related to the event will have a high frequency. Analyzing the most frequent terms in social posts is crucial in many practical applications [1, 6, 7]. To illustrate, assume we would like to mine and analyze the social media data about points of interests (interesting places) to give visitors a panoramic view of outstanding events related to the location that they are interested in. When tourists visit a place, they are not only interested in recent events [3], but they also would like to explore highlight events and features that have occurred in the place's ambiguously-defined "past". Particularly for tourists with little prior knowledge of a place, defining the boundaries of the minimally-queryable-yet-equally-relevant past is much more difficult and offers less chance for serendipity than simply expressing a greater interest towards one event A that is more recent than another event B, all else being equal between them. Considering the entire time frame of events but decaying event relevance based on recency produces results that exhibit both completeness and freshness.

Email addresses: lanfict@gmail.com (Thu-Lan Dam), schester@uvic.ca (Sean Chester), noervaag@ntnu.no (Kjetil Nørvåg), quang.huy.duong@ntnu.no (Quang-Huy Duong)

Typically, detecting trending terms consists of finding the k terms used most often in social posts within a user-specified spatio-temporal range. However, as Fig. 1 illustrates, this straight-forward frequency-based approach has its limitations. The figure shows a tag cloud created based on the thirty most popular terms from a subset of tweets from Oslo and London that were posted during the week between Christmas and New Year's Eve. Although some temporally-specific terms, such as "jul", "christmas" and "star" are visible, they are limited and vastly overshadowed by always-frequent terms, such as "oslo", "london", "norway", "uk", "great", and "photo." One approach is to normalise the term counts by their overall frequency (e.g., [8]) to discard always-frequent terms, but this biases towards always-infrequent terms. Another approach is to manipulate the temporal range, but this may yield *stale* (if the range is expanded) or *incomplete* (if it is shrunk) results.

To circumvent these challenges, this work introduces and studies a query operator that applies a *decaying, time weighted* frequency term; this assigns an exponentially higher score to terms posted most recently. The effect is visible in Fig. 1 (right): we queried Oslo and London within a time window from 2015-12-25 to 2016-01-01 (New Year's Eve), but used decay factors on the term scores to give the newer terms more weight than the old terms; the result gives much higher weight to the already present juletide terms, but also includes many other specific festive terms, such as "2015-12-25", "family", "feliz", and "jesus".

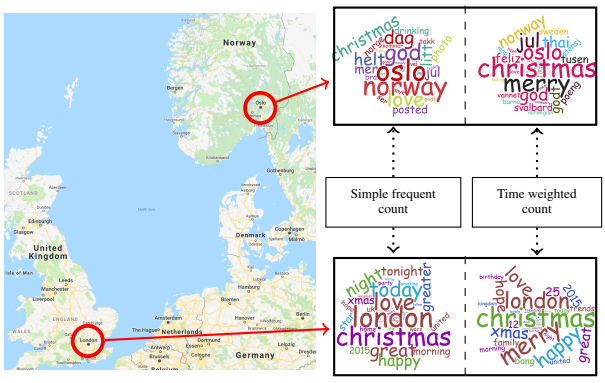


Figure 1: The 30 most common terms around time window New Year's Eve using a spatio-temporal range (left) in Oslo (top) and London (bottom) or an exponentially decaying freshness function (right), as in this paper.

Investigating decay factors as weighing mechanism for term querying from spatio-temporal textual streams is an interesting challenge. It is computationally more challenging than using a temporal range, because evaluating the score of a term: a) is dependent on when the query is issued; and b) may involve arbitrarily old social posts. A straightforward method may be to use a spatio-textual index to retrieve spatially relevant social posts, and then evaluate the score of each retrieved term. However, this is expensive because the collection of social posts that satisfy the spatial constraints could be huge. To compound the challenge, existing approaches for evaluating straight-forward frequencies within a spatio-temporal range assume a static, disk-based setting [1], which ignores that this spatio-temporal-textual data arrives at a rate of thousands of records per second; otherwise, they assume a streaming data model [2, 3, 4, 5], which necessitates an explicit temporal range.

We thus propose a new indexing technique to efficiently update and query social posts to retrieve fresh, complete, spatio-temporally relevant terms. It is based on a *time-weighted term list* that maintains terms sorted by occurrence relevant to a landmark epoch. This allows us to process a query with classic aggregation algorithms, such as TA [9] or BPA [10]. Our idea is sufficiently general that it can be applied to various spatial indexing structures. We evaluate our technique, applied to both the space partitioning quad-tree and the more ubiquitous R-tree. While both methods outperform a competitive baseline by several orders of magnitude, the quad-tree version achieves much better performance than the R-tree by minimizing the number of time-weighted term lists that need to be aggregated. To this end, our main contributions are:

- We introduce, for the first time in the literature, a location-based time-decaying query to retrieve recently frequent terms within a user specified region of interest, and we propose both exact and approximate algorithms to address it efficiently;
- We introduce the time-weighted term list structure (TwTL) to enable both quad-trees and R-trees to index social post streams;
- We demonstrate how to support fast digestion of social streams with a batch insertion of simultaneous Morton encoding and time-weighted frequency pre-calculation;
- We perform an extensive experimental evaluation on both real-world and synthetic data to evaluate query response performance and index cost. The results show that our methods are highly efficient in terms of query response time, accuracy and scalability.

Reproducibility: The source code used in this paper is publicly available on GitHub.[1]

The rest of this paper is organized as follows. In Section 2, we briefly present related work. In Section 3, we define the problem and introduce preliminaries. In Section 4, we present the index construction and query operations. In Section 5, we report and discuss the experimental results. Finally, in Section 6, we conclude the paper.

2. Related Work

Top-k spatial keyword query. The task of a top-k spatial keyword query is to retrieve the k most relevant spatio-textual objects by considering both proximity to the query location and textual similarity to the query keywords [11, 12]. In order to organize spatial and textual information, many hybrid index structures have been proposed. The spatial indices can be categorized into three main categories: tree based (R-tree, quad-tree) [13, 14, 15, 16], grid based [17, 18], and space filling curve based [19]. In contrast, most textual indices use an inverted file structure. However, these works aim at retrieving top-k geo-textual objects containing queried keywords, which is different from the task of retrieving top-k terms.

Top-k spatio-temporal term query. Given a user-specified spatio-temporal region, this query finds the k most frequent terms in the posts within this query region [2, 3, 1]. GeoTrend [3] computes the top-k most trending keywords that are posted within an arbitrary spatial region and during the last T time units. This method uses a spatial grid where each cell maintains a materialized list of top-k trending keywords that appear within the cell spatial boundaries, where the time period depends on the size of system's main memory. To generate the query results, the lists of top-k trending keywords in relevant cells are combined. However, each index cell is equipped with an expiration module; therefore, any data which is older than T is considered to have expired and is thrown out. Furthermore, GeoTrend calculates the trend of keywords by a trend line slope formula, where the last T time units are divided into N equal time intervals. The higher values of N, the more accuracy but the more expensive in computation and memory; in [3] the value of N was relatively small (N=8 counters per hash entry).

[1] `https://github.com/topkRFT/KYP` (To appear after paper acceptance.)

AFIA [2] uses a multi-layer grid-based index structure where each cell maintains the k+1 most frequent terms. Given that a top-k algorithm may need more than k entries from each list, there is no guarantee that the resulting top-k terms are 100% accurate. Instead, its output is divided into two subsets, one with X terms (where $X \leq k$) that are guaranteed to be in top-k, and the rest $k-X$ terms that are approximate top-k terms. AFIA only counts simple frequency, so it does not support answering trending queries. Furthermore, AFIA returns top-k frequent terms at different locations in a recent time interval, which is different from our problem: our problem considers the entire time frame.

Recently, kFST [1] presented a basic analytic query on geo-tagged data: given a spatio-temporal region, the objective is to return the most frequent terms in social posts in that region. kFST employs an R-tree augmented by top-k sorted term lists (STLs). Then a top-k aggregation algorithm (e.g., RA, NRA [9]) is applied on the STLs of nodes that intersect with the query region. However, kFST only counts simple frequency and operates on static data which is less relevant in social networks.

Content based Publish/Subscribe. Publish/subscribe systems support a number of *subscribers* and allow them to continuously receive messages/objects relevant to their subscriptions from *publishers* [20, 6, 21], where the subscriber specified region and the subscriber specified keywords in a subscription query perform as Boolean filters. However, a subscriber may receive very few or a huge volume of matching objects, which is depending on the specified query region and the specified query keywords. Therefore, top-k publish/subscribe systems were introduced which return a subscriber only top-k messages/objects that are ranked based on score functions (i.e., keyword relevance, location and freshness) [22, 23, 7]. However, these works continuously feeds subscribers with relevant geo-textual objects, which is different from the task of retrieving top-k terms. Lastly, there does not exist a reasonable way to adopt these systems to handle our query.

Frequent item counting. There have been numerous studies to find frequent items from a data stream with a sliding window model (i.e., valid frequent items are restricted to those in the current window) [24, 25] or time decay model (i.e., the weight of the received items is decreased over time, and the frequent items are then computed based on time decayed counts) [26, 27, 28]. The time decay model is more desirable because it not only considers every item that arrives but also incorporates the freshness of items. A great deal of work

has been proposed to find frequent items from data streams with the first model, which can be classified into two main groups: counter-based [24, 29] and sketch-based [30, 31]. The representative algorithms for this model are Space Saving [24], Lossy Counting [29], Count-Min Sketch [30] and FSS [31]. The counter-based algorithms [24, 29] monitor a subset of items from the stream by maintaining a set of counters to track the frequent items over the subset. Sketch-based algorithms [30, 31] use a set of array counters to estimate the frequency of items. However, differently from the counter-based techniques, each item is projected into a set of corresponding sketches by some hash functions. The frequency of an item is estimated from the counter of its corresponding sketches.

Many efficient algorithms also have been introduced for finding frequent items in a time decayed stream [26, 27, 32, 28]. These methods utilize a decay function, which takes information about an item and returns a weight for it. There are two ways of measuring items' weight: as a function of its age, or as an amount of time between the arrival of item and a fixed point [26]. The former is referred to as *backward decay*, as we measure back from the current time to the item's timestamp. The latter is termed *forward decay*, as we look forward in time from the stream to see the item. The most commonly used decay functions are exponential decay and polynomial decay [27, 32, 28]. TwMinSwap [27, 32] is a highly efficient algorithm for this model.

3. Problem Definition

The objective of this paper is to efficiently retrieve terms that are trending in a given spatial region, without explicitly defining a temporal window. In this section, we formally define this *top-k recently frequent terms* (*kRFT*) query (Problem 1).

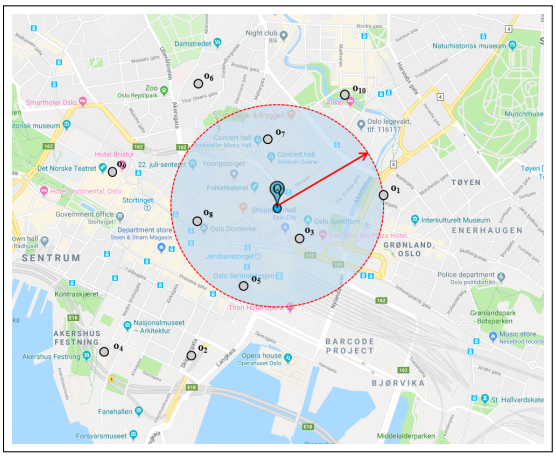
Definition 1 (Social Post)**.** *A social post is a quadruple* $o = \langle o.id, o.Time, o.Loc, o.Terms \rangle$*, where* $o.id$ *is a unique id;* $o.Time$ *is the time of creation;* $o.Loc$ *is the 2d location of the post (latitude and longitude); and* $o.Terms = \{t_1, t_2, \dots\}$ *is the multiset of the post's terms.*

Common examples of social posts are geo-tagged tweets in Twitter and geo-tagged news in news portals. Given a stream of social posts, $\mathcal{D} = \{o_1, o_2, \dots, o_n\}$, the *vocabulary* of $\mathcal{D}$ is the set of all unique terms that occur in at least one post: $\mathcal{V}_\mathcal{D} = \cup_{o \in \mathcal{D}} o.Terms$.

Example 1. *A social post stream,* $\mathcal{D} = \{o_1, o_2, \dots, o_{10}\}$*, is illustrated in Fig. 2. Each* o_i *consists of a creation time and term list (Fig. 2a) as well as a physical location (Fig. 2b). Together, social posts* o_1*–*o_{10} *define a vocabulary of 10 unique terms,* $\mathcal{V}_\mathcal{D} = \{t_1, \dots, t_{10}\}$*. Fig. 2b also illustrates a region of interest* R_Q*, the blue circle of radius 500m centered at the pin (Oslo's epicentre).*

(a) Textual and temporal information

Id	Time of creation	Terms
o_1	10:42 Jun 1, 2018	$\{t_1, t_2, t_4\}$
o_2	09:40 Jun 4, 2018	$\{t_2, t_2, t_6\}$
o_3	19:02 Jun 2, 2018	$\{t_1, t_3, t_4, t_7\}$
o_4	19:22 Jun 4, 2018	$\{t_3, t_6, t_8, t_8\}$
o_5	17:19 Jun 6, 2018	$\{t_4, t_5, t_7, t_9\}$
o_6	21:12 Jun 4, 2018	$\{t_1, t_1, t_2, t_5, t_8, t_{10}\}$
o_7	23:57 Jun 3, 2018	$\{t_4, t_5, t_6, t_9\}$
o_8	10:02 Jun 4, 2018	$\{t_2, t_4, t_6, t_7\}$
o_9	18:31 Jun 1, 2018	$\{t_6, t_6, t_8, t_{10}\}$
o_{10}	22:19 Jun 5, 2018	$\{t_5, t_7, t_9\}$



(b) Spatial information

Figure 2: Example instance of the *kRFT* query. (a) Each social post is associated with a timestamp and multiset of terms. (b) The blue circle delineates the five social posts that occurred within the user-specified distance of Oslo's centre (the pin). One wants to rank t_1–t_{10} by Eq. 2, using only within-range posts: $\{o_1, o_3, o_5, o_7, o_8\}$.

For the sake of brevity, in the rest of the paper, we use "post" instead of "social post". As with previous literature [1, 2, 3, 4], we are interested in the frequency with which a term appears in a region:

Definition 2 (Regional frequency of a term)**.** *Let* $f_{o_i}(t)$ *denote how often term* t *occurs in* $o_i.Terms$*. Then, given a region of interest* R_Q *and a social stream* $\mathcal{D}$*, the frequency of term* $t \in \mathcal{V}_\mathcal{D}$ *in* R_Q *is:*

$$f_{R_Q}(t) = \sum_{o_i \in R_Q} f_{o_i}(t). \quad (1)$$

Unlike previous literature, however, we score the relevance of a term by applying a decaying weight to the frequency terms:

Definition 3 (Regional time-weighted frequency of a term)**.** *Let t_{curr} denote the time of a query and $0 < \alpha \leq 1$ be a decay rate. Then, given a region R_Q and a social stream $\mathcal{D}$, the time-weighted frequency of term $t \in \mathcal{V}_\mathcal{D}$ in R_Q is:*

$$wf_{R_Q}(t) = \sum_{o_i \in R_Q} f_{o_i}(t) \times \alpha^{t_{curr} - o_i.Time}. \quad (2)$$

The exponential time-weighting of frequencies gives a much higher emphasis to terms that occur in newer posts. A small value of α gives lower weight to older posts, whereas $\alpha = 1$ gives equal weight to everything, irrespective of creation time. Using time-weighted frequency in lieu of temporal ranges preserves both *completeness* (all terms and posts are considered) and *freshness* (recency is more important). This gives rise to the problem studied here, which is to retrieve the k terms from a social stream that maximize *time-weighted frequency*, subject to geographic constraints:

Problem 1 (top-k Recently Frequent Terms Query (kRFT))**.**

Given: *a stream of posts $\mathcal{D}$, a user-specified region of interest R_Q (a centre point and a radius), and an output size k*

Retrieve: *the k terms $t \in \mathcal{V}_\mathcal{D}$ that best maximize $wf_{R_Q}(t)$ at the querying time.*

Example 2. *Consider the example in Fig. 2, the user-specified region of interest R_Q is the blue circle that includes the posts $\{o_1, o_3, o_5, o_7, o_8\}$, assume that $\alpha = 0.9$, $k = 4$, the query date time is Jun.06.2018, and time unit is one day. The regional time-weighted frequency of terms from the posts related to the region of interest are calculated by Eq. 2, i.e., $wf_{R_Q}(t_1) = 1 \times 0.9^{(6-1)} + 1 \times 0.9^{(6-2)} = 0.59 + 0.66 = 1.25$ because t_1 occurred in o_1 and o_3. Calculate in the similar way for the remaining terms we have: {t_1: 1.25, t_2: 1.4, t_3: 0.66, t_4: 3.79, t_5: 1.73, t_6: 1.54, t_7: 2.47, t_9: 1.73}, where the number after the colon for each term indicates its regional time-weighted frequency. Then the resulting top-4 terms of kRFT are {t_4: 3.79, t_7: 2.47, t_5: 1.73, t_9: 1.73}. If only simple frequency is counted, then the returned top-4 frequent terms are {t_4: 5, t_7: 3, t_1: 2, t_2: 2}, which is different from the result of kRFT and includes the very old terms, e.g., t_1; whereas the returned result of kRFT includes both the long-life permanent frequent terms (e.g., t_4) as well as new frequent terms (e.g., t_9).*

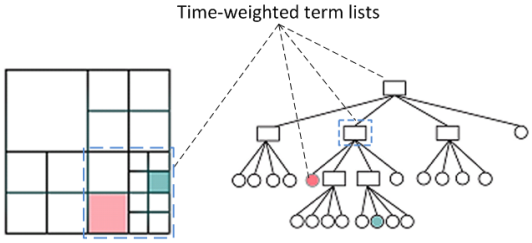


Figure 3: Structure of the hybrid quad-tree.

4. Proposed Method

We aim at reporting the top-k most recently frequent terms from a dynamic collection of posts $\mathcal{D}$ for a specific region of interest. In general, $\mathcal{D}$ is treated as a dynamic data warehouse where posts are continuously inserted, and we search over a subset of posts within a specific region to return the top-k time-weighted frequent terms. In order to support efficient insertion of new data and querying, we propose organizing posts in a spatial index and annotating the index nodes with precomputed time-weighted frequencies of the constituent terms. The details of our approach are described in the following subsections.

4.1. Index Structure

Variants of the R-tree [33, 34, 35, 36] are used ubiquitously to support spatio-temporal queries [1, 11, 15]. However, R-trees are not well suited for rapid content streams and spatio-temporal aggregate queries due to their poor update performance [37]. Another indexing alternative is the quad-tree [38, 39], a space partitioning tree data structure in which d-dimensional space is recursively subdivided into 2^d regions. Due to its simplicity and regularity, quad-trees have also been widely adopted in many applications, especially when there is no strict requirement that the tree is balanced. Therefore, we adopt quad-tree structure to index posts.

Further, we organize posts into nodes to support: (*i*) efficiently identifying the exact set of posts that fall within the region of interest; and (*ii*) balancing the number of posts among the child nodes. In addition, we precalculate the time-weighted frequency of terms which

are stored in a Time-weighted Term List (TwTL) for every node (cf., Fig. 3). Then, the final result set is computed by retrieving and combining the pre-calculated TwTL associated with the set of posts belonging to the search region. The hybrid spatial index is built online and new posts are optionally inserted in batch. Fig. 3 shows the proposed quad-tree structure; the TwTL and tree construction process will be presented in the following sections.

4.2. Time-weighted Term List

We introduce the Time-weighted Term List (TwTL) structure, which stores term statistics to accelerate query processing in this section.

Definition 4 (Time-weighted Term List)**.** *Each TwTL element is a 2-tuple describing a unique term, t, including: the term itself and its accumulated time-weighted frequency.*

However, it is not obvious how to efficiently calculate time-weighted frequency by Eq. 2 given that t_{curr} is not fixed: compared to simple frequency counting, the time-weighted frequency of a term is dynamic. It is infeasible to maintain up-to-date time-weighted frequency by re-computing all statistics whenever posts are added or at query time. Further, we need a simple but effective mechanism to maintain streams of posts. To address this challenge, we introduce an efficient strategy for calculating the time-weighted frequency without re-computation, in line with the *forward decay model* of [26]. Let ts_0 be the earliest timestamp observed in the stream and $te_{o_i} = o_i.Time - ts_0$ is the age of o_i since the start monitoring time ts_0. Then we have the following property:

Property 1. *Consider generally the whole world region and its social stream $\mathcal{D}$, the time-weighted frequency of term $t \in \mathcal{V}_\mathcal{D}$ by Eq. 2 is:*

$$wf(t) = \sum_{o_i \in \mathcal{D}} f_{o_i}(t) \times \alpha^{(t_{curr} - o_i.Time)}. \tag{3}$$

And

$$wf(t) \propto \sum_{o_i \in \mathcal{D}} \frac{f_{o_i}(t)}{\alpha^{(o_i.Time - ts_0)}}, \tag{4}$$

where $0 < \alpha \leq 1$, and ts_0 is the earliest observed time.

Proof. We can rewrite Eq. 3 as:

$$wf(t) = \sum_{o_i \in \mathcal{D}} f_{o_i}(t) \times \alpha^{(t_{curr} - ts_0)} \times \alpha^{(ts_0 - o_i.Time)}. \tag{5}$$

Table 1: Example Time-weighted Term List (TwTL).

Term	t_1	t_2	t_3	t_4	t_5	...
wf	4.85	6.48	2.48	6.41	5.82	

Observe that the middle term $\alpha^{(t_{curr} - ts_0)}$ does not depend on the summation variable o_i; so, we can extract it from the summation:

$$\frac{wf(t)}{\alpha^{(t_{curr} - ts_0)}} = \sum_{o_i \in \mathcal{D}} f_{o_i}(t) \times \alpha^{(ts_0 - o_i.Time)}. \tag{6}$$

Moreover, as t_{curr} is fixed at query time, the extracted term $\alpha^{(t_{curr} - ts_0)}$ is just a constant weight for all terms. Thus, (by multiplying the exponent of the remaining α-term by -1 and taking its reciprocal) we have:

$$wf(t) \propto \sum_{o_i \in \mathcal{D}} \frac{f_{o_i}(t)}{\alpha^{(o_i.Time - ts_0)}}.$$

□

Example 3. *Table 1 shows an example TwTL structure where $ts_0 = o_1.Time$, $\alpha = 0.9$, and the time unit is one day. Term t_1 appears in o_1 once, in o_3 once, and in o_6 twice. For all other o_i, $wf(t_1) = f_{o_i}(t_1) = 0$. By Eq. 4, $wf(t_1) = \frac{1}{0.9^{1-1}} + \frac{1}{0.9^{2-1}} + \frac{2}{0.9^{4-1}} = 1 + 1.11 + 2.74 = 4.85$. Similarly, $wf(t_2) = 1 + 2.74 + 1.37 + 1.37 = 6.48$.*

Then each node of the index tree is associated with one TwTL, which summarizes all terms occurring therein and its children. This list is updated when inserting posts into the tree. The next subsection will present how TwTL is augmented to the spatial index.

4.3. Index Insertion in the Hybrid Quad-tree

Inserting an individual post o_i into a spatial index is typically performed by traversing the tree from its root node to find the leaf node that contains the location of o_i [16, 4], i.e., first executing a query for o_i. However, this approach is not sufficiently responsive for real-time stream processing with high arrival rates, because it incurs an additional load of thousands of queries per second. In order to support a high arrival rate, we employ an efficient bulk insertion technique that compared to the traditional way of inserting individual posts can reduce a large number of comparison operations for term locations with spatial node boundaries [2].

[2] Bulk loading is only supported for the quad-tree, which, unlike an R-tree, never has to rebalance after a node split.

The main idea is to insert posts in batches. In particular, we buffer incoming posts in a memory buffer B and build a map of their terms at the same time. Furthermore, the location of each post is converted to a Z-order code (also known as Morton order or Morton code) [40] in order to enable a fast split into ranges when bulk inserting into the quadrants later. The text is split into terms and stop words are removed on-the-fly to ensure they do not dominate the top-k results. The buffer B has two parts: a *zlist* containing z-order codes of incoming posts, and a term map M summarizing terms from these posts, each entry of M has the form of $\{t, [\langle o_{id}, f_{o_{id}}(t)\rangle, \dots]\}$, where o_{id} is the id of post containing term t and $f_{o_{id}}(t)$ is the number of times t appears in o_{id}. Before inserting, we sort *B.zlist* in ascending order of its elements. Then, the bulk insertion is performed to insert B into the hybrid quad-tree, which consists of two steps: (1) traversing the tree with batches of terms, and while traversing, (2) the terms are merged into its corresponding nodes. The details of these steps are as follows.

Quad-tree traversal. The terms in map M are first inserted into the root node N of the tree as described in *Node insertion* below. If N is a non-leaf node, we split the buffer B into sub-buffers with respect to the quadrants of N accordingly. Each sub-buffer encloses a subset of z-orders and a term map that corresponds to one of the quadrants. Then, the same insertion process is applied to each of the quadrants using the corresponding buffer. If N is a leaf node and its contents does not exceed the predefined maximum node capacity, we store the arrived posts and merge the contents of B to N. Otherwise, we create four children of N and employ the same process as on the non-leaf node above.

Node insertion. For each newly arrived term, if there is no corresponding element in the TwTL of the node N, *N.TwTL*, we initialize an element for this term. Then, entries of a term in M are concatenated to the corresponding term in *N.TwTL*, and the time-weighted frequency of the arrived term in *N.TwTL* is incremented by a value calculated by Eq. 4, where o_i is the new post.

Algorithm 1 shows the pseudocode of our index construction method where the whole TwTLs are stored in all nodes, which is named *Full-TwTL*. Employing the z-order and sorting the *zlist* in ascending order brings several benefits to the bulk insertion. First, splitting the *zlist* of buffer B into several sub-lists according to the z-ranges of quadrants is easy. Second, posts having locations which are near others are inserted into tree nodes that are also nearby in the tree. This, in turn, would improve the query efficiency later.

Algorithm 1: Batch Quad-Tree Construction

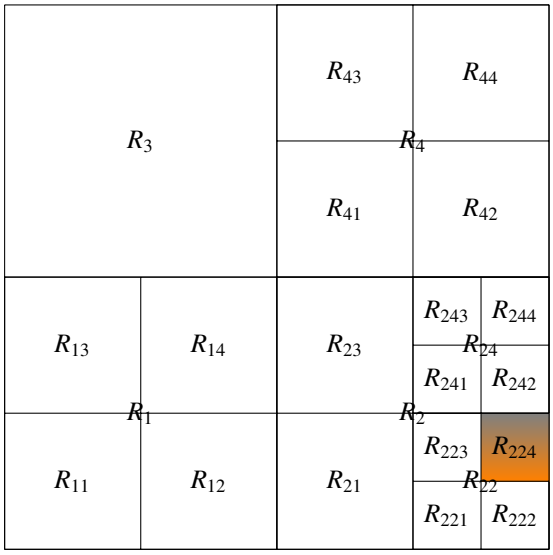
```
Input: A stream of posts D.
Output: Quad-tree QT.
Initialize tree QT and buffer B if needed
while buffer B is not full do
    Read o_i from D
    Calculate z-order from o_i.Loc and add to B.zlist
    Extract terms from o_i.Terms and add to B.M
Sort B.zlist in ascending order
N ← root of QT
Call InsertTree(N, B)

Function InsertTree(N, B)
    Merge term map B.M to N.TwTL
    if N is internal node then
        Split B into sub-buffers w.r.t. N.quadrants
        Call InsertTree(N.quadrant, sub-buffer)
          accordingly
    else
        if N is leaf node And N contains B.zlist
          then
            if (N.capacity + size of
              B.zlist) ≤ MaxCapacity then
                Add B to N;
            else
                Merge data of N to B
                Create N.quadrants
                Split B into sub-buffers w.r.t.
                  N.quadrants
                Call InsertTree(N.quadrant,
                  sub-buffer) accordingly
```

Illustrated example. Figure 4 illustrates in detail how we traverse the Quadtree and insert data to a node in the tree. Given a spatial region as in Fig. 4 (top-left), indexed using an Quadtree as presented in Fig. 4 (bottom). The detailed steps of the process are presented in Fig. 4 (top-right). For the sake of simplicity, in the example, we use $\alpha = 1.0$ and the buffer size B is set to 1 (size of B.M). The algorithm is an approximate method and value of K is set to 5. The values on edge of the tree are Morton ranges (from-to) of the child region.

4.4. TopK Time-weighted Term List

In some cases, the Full-TwTL approach described above requires a lot of memory, especially for the TwTL at higher-level nodes, which may contain the whole vocabulary $\mathcal{V}$. This would also degrade the query perfor-



Step 1: new post $o_2\{t_2\ t_2\ t_6\}$ with id=2 arrives at the location R_{224}

Step 2: calculate the z-order of o_2, we have $o_2.z - order = 23$, then add this value to *zlist* of B: B.*zlist*.add(23)
- Extract terms of o_2 with weights, we have 2 terms $\{t_2, t_6\}$, and add these terms to B.M. So, we have B.M = $\{\{t_2,[2,2]\}, \{t_6,[2,1]\}\}$

Step 3: Insert data in buffer B to the Quadtree:
- Tree traversal: start from the *Root*, traverse down to the corresponding order. On the tree, the branching path from the *Root* to the region with z-order value 23 is in orange color.
- Node insertion: after the tree traversal, we found node R_{224} that contains post $o_2\{t_2\ t_2\ t_6\}$, data will be inserted to this node. For simplicity, we assume the capacity of the node does not yet reach its maximum setting, the following steps are processed:
+ Store post o_2 in the post list of node R_{224}
+ Update the new term data in B.M with the list $TwTL$ of R_{224}. For instance, if $R_{224}.TwTL$ = {[t_1:1], [t_6:1]}, after the update with k = 5, $R_{224}.TwTL$ will be {[t_1:1], [t_2:2], [t_6:2]}

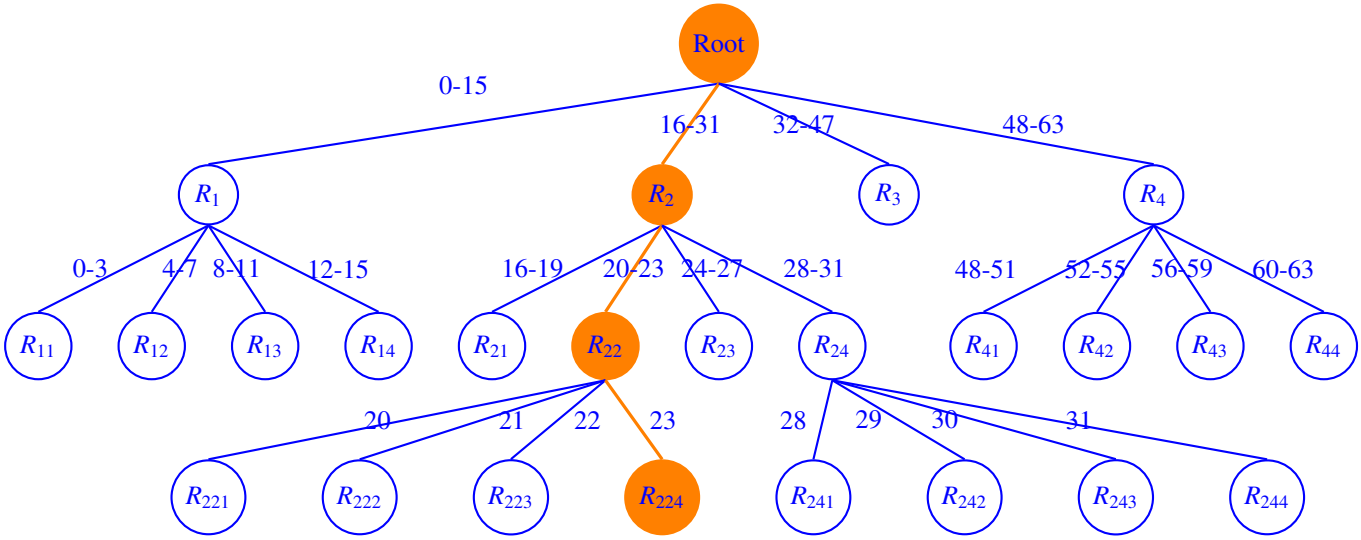


Figure 4: Example of Quadtree traversal and node Insertion.

mance because the produced results are aggregated from several corresponding TwTLs, which contain a large number of terms. Therefore, we introduce an approximate approach named *TopK-TwTL*, in which only the K terms (K counters) having highest time-weighted frequency are maintained at each node. Its index is created similar to the Full-TwTL approach, except that in the *node insertion* step, while merging new arrival terms in map M to the TwTL list stored in node N, we calculate and maintain only the K terms with the highest time-weighted frequency in this node.

Our framework is a time-aware-based approach. In order to efficiently calculate and maintain the list of Top-K time-weighted frequent terms we follow the spirit of TwMinSwap [27]. However, TwMinSwap has to recalculate frequencies of the currently monitoring items, whereas our method does not require this step. The reason is that we employ the proposed equation Eq. 4 to calculate the time-weighted frequencies of terms. Further, we start monitoring a new arrival term t with the initial count as its time-weighted frequency instead of 1. Lastly, we observe that the majority of terms in post stream are infrequent and many of them have the same frequency, therefore we follow a flexible top-k approach, i.e., the TopK-TwTL may contain more than K counters (monitored terms) when some terms have the

Algorithm 2: Construct TopK-TwTL

Input: A map of terms M, the number of counters K, a decay parameter α.
Output: Top-K time-weighted frequent terms $TopK$.

```
1  TopK ← currently monitored terms
2  for each term t from M do
3      Calculate the added wf of term t from M by Eq. 4 using α
4      if t ∈ TopK then wf_t ← wf_t + wf
5      else
6          if |TopK| < K then
7              TopK ← TopK ∪ {t}
8              wf_t ← wf
9          else
10             v* ← argmin_{v∈TopK} wf_v
11             if wf_{v*} < wf then
12                 TopK ← TopK\{v*} ∪ {t}
13                 wf_t ← wf
14             if wf_{v*} = wf then
15                 TopK ← TopK ∪ {t}
16                 wf_t ← wf
17 return TopK
```

same weighted frequency. The detailed steps of the improved algorithm are described in Algorithm 2. The index structure is built in the same way as in Algorithm 1, except that lines 10 and 19 are replaced by Algorithm 2 when merging information of terms in map M to the TwTL stored in this node to maintain only the top-K terms. Furthermore, at a specific time, the depth of a quad-tree can not be deeper due to the amount of data is huge. Therefore, when the tree reaches a maximum depth, instead of splitting the leaf node we replace a number of posts stored in the current leaf node by the new posts in the buffer B. This may affect the accuracy when the search region is not fully contained in a leaf node. However, the TopK-TwTL still provides a high accuracy, as will be demonstrated in the experimental result section.

4.5. Query Processing

Given a spatial index with pre-calculated time-weighted term frequencies, queries can be executed as per Algorithm 3. First, we find the exact set of tree nodes SN whose TwTLs involve the query region R_Q such that it minimizes the number of nodes in SN starting from the root node of the tree (Algorithm 4). Then,

Algorithm 3: kRFT(T, R_Q, k)

Input: Tree T, region of interest R_Q, number of terms k.
Output: Top-k highest time-weighted frequency terms in R_Q.

```
1  Initialize Tabk = ∅, SN = ∅
2  Calculate z-range = [z_min, z_max] of points in R_Q
3  SN ← FindNodes(T.root, z-range)
4  Tabk = mBPA2-TwTL(SN, k) /
       mBPA2-TopK-TwTL(SN, k)
5  return Tabk
```

the second step finds top-k keywords in R_Q by aggregating the time-weighted term lists that are maintained in SN' nodes (Algorithm 5). Note that the exact method Full-TwTL aggregates the values from the Full-TwTLs stored in corresponding nodes, while the approximate method TopK-TwTL uses the TopK lists in the nodes. Further, we follow the spirit of the best position algorithm [10], which executes top-k queries more efficiently than TA [9] by avoiding re-accessing data items via random access. However, there are several improvements: (1) the involved lists must be sorted first (line 4), and (2) because the lengths of input lists are different, we have to check the availability of the current term (line 10).

5. Experimental evaluation

This section presents an experimental evaluation of our proposed time-weighted term list (TwTL) and indexing methods for solving *kRFT*, where we will evaluate the algorithms both in terms of query performance (Section 5.2), and the construction/maintenance cost of the relevant indexes (Section 5.3). But first, we describe the computational environment, implementations, datasets and query selection (Section 5.1).

5.1. Experimental Methodology

Environment. The experiments were run on a server with dual 14-core 2.00 GHz Intel Xeon E5-2683 v3 (Haswell) CPUs, with 192 GB DDR4 RAM, a 7.3 TB SATA disk, and Ubuntu 18.04. The relevant datasets and all data structures are always memory-resident at query time, and caches are warm.

Implemented algorithms. Recall from Section 2 that there are no relevant methods in the literature to solve the *kRFT* query: existing algorithms for returning top-k frequent terms assume static data, require a pre-defined

Algorithm 4: FindNodes(N, *z-range*)

Input: A tree node N and *z-range* of query region R_Q.
Output: Set of satisfied nodes SN and its TwTLs/TopK-TwTLs.

```
if N is a leaf node then
    if z-range ⊂ N.z-range then
        return N and its TwTL/TopK-TwTL
    else return ∅
if N.z-range is totally enclosed in z-range then
    return N and its TwTL/TopK-TwTL
else
    SN ← ∅
    Decompose z-range into sub-z-ranges according to N.quadrants
    for each quadrant in N.quadrants do
        if quadrant.z-range ⊇ a sub-z-range in sub-z-ranges then
            SN ← SN ∪ FindNodes(quadrant, sub-z-range)
return SN
```

Algorithm 5: mBPA2-TwTL(SN, k)

Input: Set of nodes SN and its TwTLs/TopK-TwTLs, the number of result terms k.
Output: k terms with highest time-weighted frequencies.

```
Initialize result set Tabk ← ∅, which can contain k elements
for each node N_i ∈ SN do
    L_i = N_i.TwTL / N_i.TopK-TwTL
    Sort L_i in ascending order of term's time-weighted frequencies
    Initialize position list P_i ← ∅ containing the positions of seen data items in L_i
    Initialize the best position bp_i ← 0
repeat
    for each list L_i ∈ SN.TwTLs / SN.TopK-TwTLs do
        t = L_i[bp_i + 1]
        if t is not null AND t ∉ P_i then
            wfreq = wf_{L_i}(t)
            P_i ← P_i ∪ bp_i + 1
            for each list L_j ∈ SN.TwTLs / SN.TopK-TwTLs do
                Do random access for t in L_j to find wf_{L_j}(t)
                wfreq ← wfreq + wf_{L_j}(t)
                Update P_j
            Tabk ← Tabk ∪ {⟨t, wfreq⟩}
            λ = Σ_{i=1}^{|SN|} wf(L_i[bp_i])
            v* ← argmin_{v∈Tabk} wf_v
until λ < wf_{v*}
return Tabk
```

temporal interval, and/or cannot straight-forwardly support a query-time-dependent decay factor on the frequency counts. Therefore, we implemented two baselines by adapting the existing methods STL-LI (Full Lists on All Nodes) and STL-li (Partial Lists on All Nodes) of kSFT [1], and named it as RTiFull and RTiTopK, respectively. STL-LI and STL-li works with static dataset and only count static simple term frequency. Both STL-LI and STL-li index all posts in an R-tree, where a sorted term list containing pairs of terms and its simple frequencies is maintained in the tree node, and STL-LI and STL-li follow the sprits of RA and NRA to produce results from the relevant term lists. Meanwhile, the adapted RTiFull and RTiTopK maintain TwTL/TopK-TwTL in its nodes, insert posts individually into its index and follow the same query processing schema as with our quad-tree versions. The R-tree implementation uses the QuadraticSplit [33] method to split nodes. The quad-tree implementation encodes nodes by Morton code (Z-order) [40] so that each quad-tree node has a unique Morton range [41]. Furthermore, we implemented a naive method, *Baselinescan*, which adopts a *spatial first* strategy: it scans the dataset to find the set of posts produced within the query region, and then naively calculates the top-k result terms from this subset. *Baselinescan* returns the exact *kRFT* result; it is thus also used as a benchmark to evaluate the accuracy of the approximate TopK-TwTL methods and the correctness of the Full-TwTL implementations. Hence, in all, we compare five algorithms: four index-enabled techniques (the two proposed methods Full-TwTL and TopK-TwTL employing quad-tree, RTiFull and RTiTopK) and a naive scan method. All algorithms were implemented in Java and are publicly available on GitHub.

Datasets. We used two datasets, named WW and CN, where WW is a real-world dataset and CN is a semi-real-world dataset. The summaries of these two datasets are shown in Table 2, while its details are described below.

WW dataset. This is the primary experimental dataset and consists of $n = 125$ million (M) geo-tagged tweets collected from the whole world using the public Twitter streaming API. Each tweet contains a timestamp, a precise longitude and latitude, and a text message containing 11 terms on average (not including stopwords). The temporal domain covers late December 2015 and early January 2016. To measure scalability, we prepared three other datasets which contain $n = 15$, 30 and 60 million tweets by extracting subsets from the $n = 125$M WW dataset, and we name these datasets 15M, 30M and 60M respectively. A subset is created by selecting the first 15, 30 or 60 million tweets from the original dataset (which is in temporal order).

CN dataset. This dataset contains approximately 30 million weibos (microblogs) posted from week 47 to week 52 (the last 6 weeks) in 2012 which is extracted from the Weiboscope Open Data[3]. The Weiboscope dataset was collected from Sina Weibo in China. However, the information about location of every weibo is not available. In order to use the dataset for our purpose, we create a geographic coordinate to each weibo as follows: we first pick a city in China randomly using a uniform distributed function, and then add a random distance to its longitude and latitude. The average text message length of the CN dataset is 21, which is longer than that of the WW dataset. We also create two other datasets which contain 10 million and 20 million weibos by extracting subsets from the CN dataset in the same way as for the WW dataset. Note that because of the way locations are created for the CN dataset, the dataset will not contain distinct hotspot locations as is the case for the WW dataset. The reason for doing it this way is to be able to study query and indexing performance for a context with less distinct hotspots than the WW dataset scenario.

Queries. For the query experiments, we generate several sets of 200 independent queries grouped in two categories: (*i*) the output size k is fixed to 10 while the query radius r is varied from 1km to 20km, in particular $r \in [1, 2, 5, 15, 20]$ (km); and (*ii*) the query radius r is fixed to 10km while the output size k is varied from 10 to 100, in particular $k \in [10, 20, 40, 60, 80, 100]$. For each query, we select a random tweet (or a weibo) from the medium sized WW dataset, $n = 30$M (or from CN dataset, $n = 20$M) and centre the query at the location of this tweet/weibo. As a result, every query will return a non-empty result (at a minimum, producing a non-zero score for all terms in that tweet/weibo), and there are 11 sets of queries in total. Each plotted data point reports the average for all 200 queries in the relevant query set.

Unless otherwise indicated in the plot or caption, the parameters assume default values as follows: the decay factor $\alpha = 0.9$ as suggested in the literature [27, 32], the output size $k = 10$, the query region radius is 10km, the node capacity is 1000, the default size of TopK-TwTL (K) is 500, the batch size is 12000 tweets/weibos, and the time unit is one day.

5.2. Query Performance

In this section, we evaluate the query performance of the algorithms.

5.2.1. Query response time

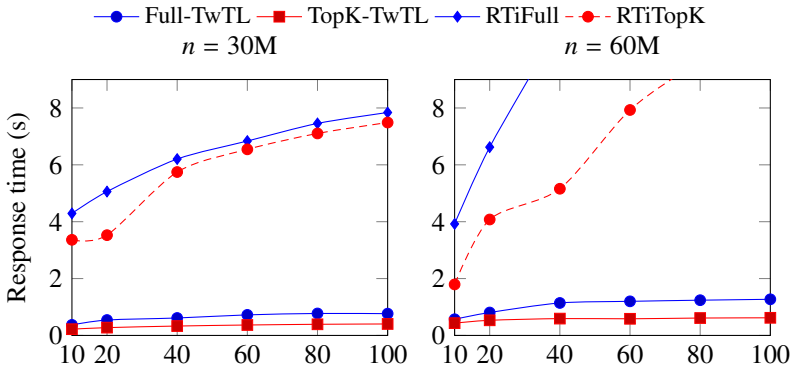
Figs. 5–6 show the average query response time for the four index-enabled methods. The plots on the top vary the output size ($k \in [10, 100]$) and those on the bottom vary the query radius ($r \in [1, 20]$ km). In particular, Fig. 5 shows the average query response time for the four index-enabled methods on the WW dataset. The plots on the left show performance at 30M tweets; those on the right, at 60M. Meanwhile, with the CN dataset, the plots on the left of Fig. 6 show performance at 20M weibos; those on the right, at 30.39M (or the whole CN dataset). The performance of *Baselinescan* is not shown, because it requires approximately 140 seconds on WW dataset and 100 seconds on CN dataset, irrespective of query parameters (k, radius); i.e., it is outperformed by 3–6 orders of magnitude and hinders the readability of the plots. Nonetheless, this indicates that specialized methods are necessary for real-time *kRFT* queries.

Overall, we can see from the plots that the quad-tree methods (Full-TwTL and TopK-TwTL) perform fastest. At the most extreme query parameters, the performance gap between the exact TwTL applied to the quad-tree versus the R-tree reaches 18.3× ($k = 100$) and 9.9× ($r = 20$). For the approximate/Top-500 TwTL, the gaps grow to 18.6× and 6.5×, respectively. At the least extreme (most probable) query parameters, these methods retain performance gaps of 11.6×, 13.6×, 15.3×, and 59.7×, respectively. In other words, the quad-tree performs substantially better at probable query values; while the gap between the quad-tree and R-tree performance shrinks as k and r increase, it is still over 6-fold when the query region encompasses roughly 1200 km^2. Thus, we conclude that, in terms of raw query performance, the TwTL is more suitable to the quad-tree and both trees are far preferable to not using a TwTL.

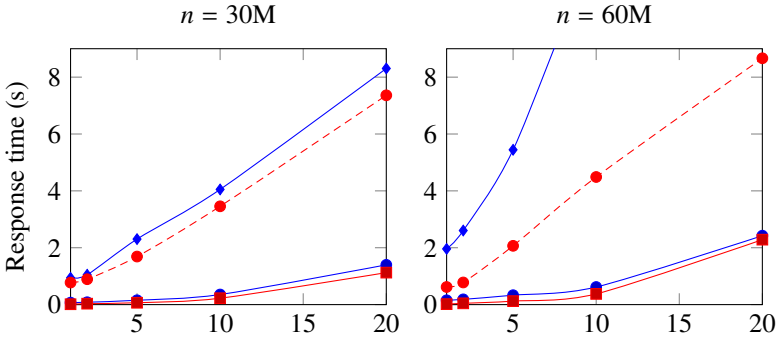
[3] `https://hub.hku.hk/cris/dataset/dataset107483` [visited August 27, 2019]

Table 2: Statistics of datasets.

Dataset	WW	CN
#Social posts	125M	30,390,646
Temporal domain	from late Dec-2015 to early Jan-2016	from week 47 to week 52, 2012
Avg. message length	11	21
$\|\mathcal{V}\|$	7,548,262	4,165,647



(a) Output Size, k.

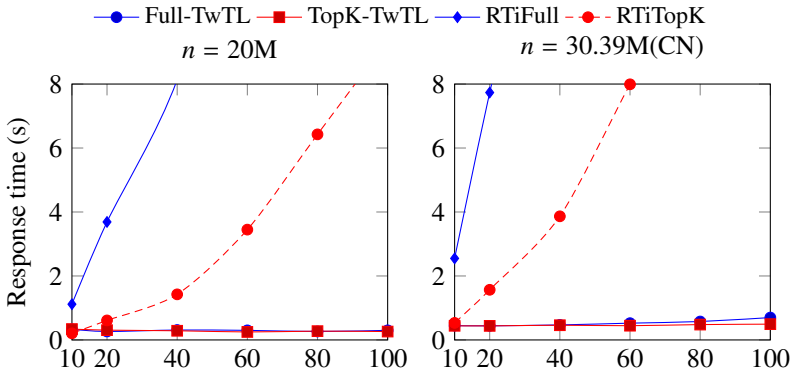


(b) Radius (km).

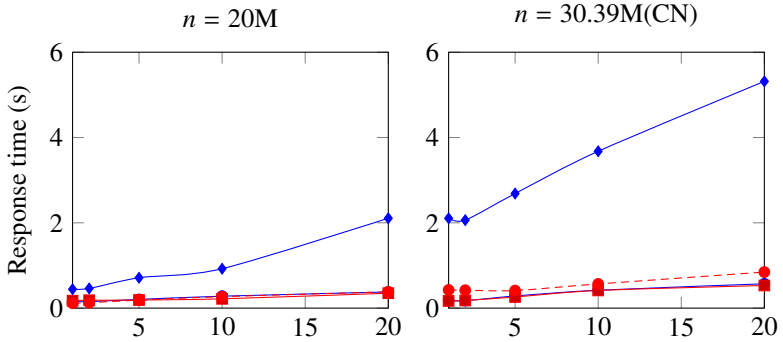
Figure 5: Query response time relative to output size k (a), radius (b), and number of posts (left vs. right) for indexed methods on WW dataset. The baseline scan (not shown) is 4–5 orders of magnitude slower.

Considering the Top-500 TwTL *approximation* in terms of Fig. 5, we see that the R-tree query response time is accelerated by 1.1–1.4× relative to the Full-TwTL, depending on the values of k and r. In contrast, the quad-tree query response time improves by 1.7–1.9× as k varies and by 1.2–5.3× as r varies. In terms of Fig. 6, the R-tree query response time is accelerated by 3.4–6.5× relative to the Full-TwTL, depending on the values of k and r. Meanwhile, the quad-tree query response time improves by 1–1.4× as k varies and by 1–1.3× as r varies; that is to say, the quad-tree response time benefits more from the approximation on the real-world dataset, but it is also more sensitive to the query radius. Unsurprisingly, the approximate methods are always faster than their exact analogues.

The overall distribution of query response time (in milliseconds) of the four index-based methods is illustrated using box plots in Fig. 7 (note logarithmic scale on Y axis). The query response time varies less for TopK-TwTL than the others, and most of its response times is less than 1 second. Furthermore, there is substantially more variation in query response time for the R-tree-based versions than that of the quad-trees. The reason for this will be further discussed in the next sub-



(a) Output Size, k.



(b) Radius (km).

Figure 6: Query response time relative to output size k (a), radius (b), and number of posts (left vs. right) for indexed methods on CN dataset. The baseline scan (not shown) is 3–6 orders of magnitude slower.

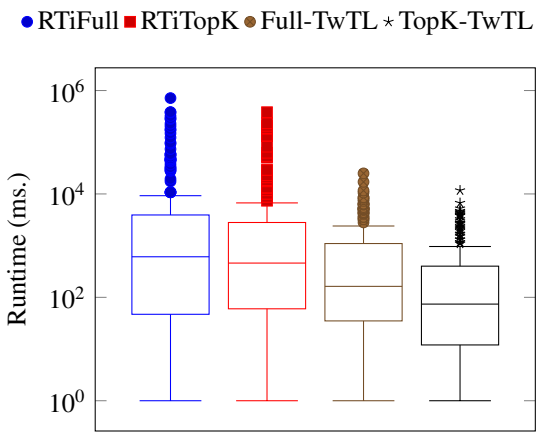


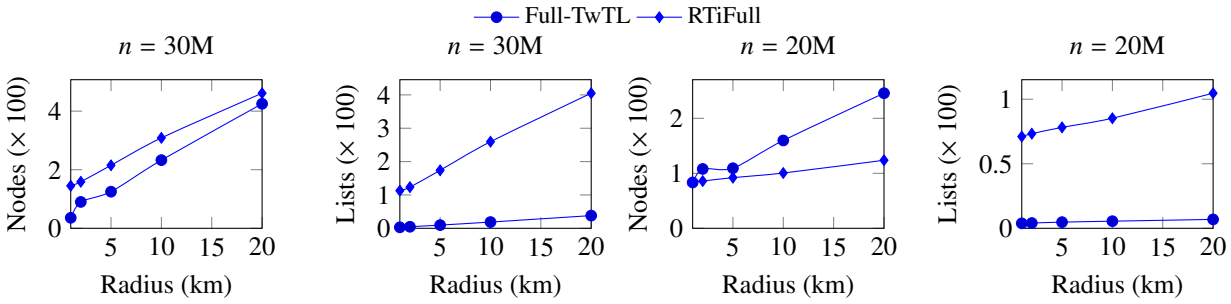
Figure 7: Box plot of query time on 60M dataset, r = 10km, k = 100.

section.

Finally, the *dataset size*, as measured by the number of posts (left plots versus right plots in Figs. 5–6), has a linearly proportionate effect on the response time of the Full-TwTL method. At smaller values of k and r, the approximate Top500-TwTL dampens the effect of a growing dataset: the two-fold increase in tweets produces only a 1.25–1.35× increase in response time. However, for the extreme values of k and r, the effect of n on the approximate method is similar to that on the exact method. Thus, we conclude that, in terms of raw query performance, the approximate Top500-TwTL can provide enhanced scalability, but mostly on the range of values that one expects to see in practice.

5.2.2. Node and TwTL accesses

Fig. 8 illuminates the trends in Section 5.2.1: the n = 30M scalability plot on WW dataset and the n = 20M scalability plot on CN dataset with respect to radius (which showed greater effect than k) are repeated, but this time measuring the number of (internal *and* leaf) nodes accessed (left) and the number of TwTL's aggregated (right). Recall from Algorithm 4 that the number of lists aggregated correlates with the number of leaf nodes, but it is possible to retrieve TwTL's at



(a) On the WW. **(b)** On the CN.

Figure 8: Number of nodes traversed (left) and number of lists aggregated (right) relative to radius on the WW and CN datasets.

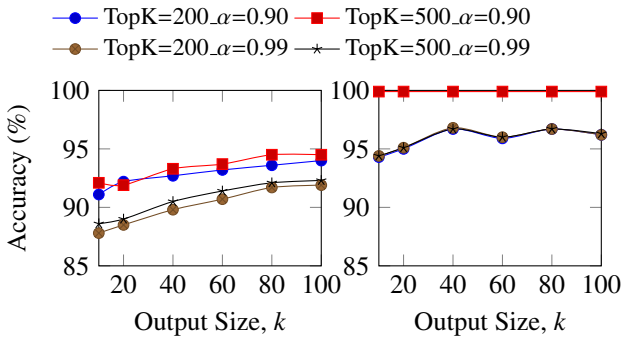
internal nodes. Broadly, we observe the same trends, but the number of lists mirrors the response time closer than node accesses do. The R-tree based index, in contrast to the quad-tree, is not a disjoint partitioning of the data space; thus, the query region may cover (intersect) many (leaf) nodes, which in turn requires processing more lists. We see similar slopes with respect to nodes accessed by each method on the WW dataset; thus, the higher cost of the R-tree is attributable to accessing more leaf nodes. With the CN dataset, the indexed R-tree is more balance and shorter in terms of tree-height than that of the quad-tree index since this dataset is more evenly distributed and does not contain hotpots as with the WW dataset, which result in the traversed nodes of R-tree index is less than that of the quad-tree.

The disjoint partitioning is what enables the quad-tree to leverage TwTLs in the internal nodes to resolve multiple ranges of posts in the query region that can be easily retrieved based on the Z-curve. With the R-tree, the same query region may include many overlapping leaf nodes, which in turn requires aggregation from a large number of term lists (TwTLs).

5.2.3. *Query accuracy*

Fig. 9 completes the query-time trade-off of using (either) approximate TopK-TwTL implementation by evaluating query accuracy. It plots accuracy as *the fraction of correct top-k terms* returned by the approximate TopK-TwTL approach compared to the naive approach *Baselinescan* (or, equivalently, either Full-TwTL implementation). We vary both the decay factor α and the approximation cut-off K as indicated in the legend, and vary the output size k along the x-axis. Note that the red-square trend line corresponds to Figs. 5–6 (top-left).

Overall, we see that Top500-TwTL achieves 90–95 % accuracy on the WW dataset and 99.9 % on the CN



(a) On WW, n = 30M. **(b)** On CN, n = 20M.

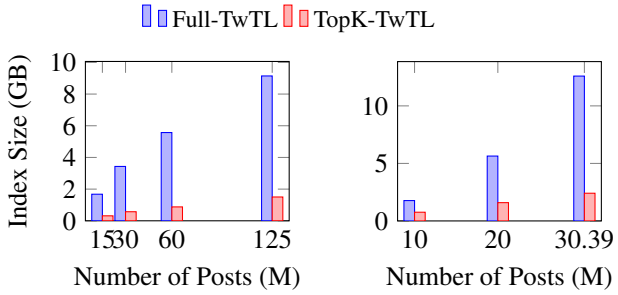
Figure 9: Query accuracy for TopK-TwTL relative to k and α.

dataset, despite only considering up to 500 terms per list. The result shows that the accuracy on CN is a little bit higher than on CN. This is explained as follows. The datasets used in this evaluation are the 30M subset from WW and 20M subset from CN. The WW subset contains 30M posts that are distributed all over the world, while the CN subset contains 20M posts over China only. The difference in the accuracy of the method in WW and CN datasets may be from the distribution density and the length of tweets. With the CN dataset, more tweets are concentrated to the same region than in WW, this could lead to largely differentiate the set of frequent terms than in a less density region as in WW dataset. A factor that also impacts to the accuracy is the vocabulary. One of the reasons that the accuracy in CN dataset is higher than in WW dataset, is because the vocabulary size of CN dataset is much smaller than the vocabulary size of WW dataset (see Table 2). Moreover, the results are very stable, despite each x-axis value corresponding to a separate, randomly generated query set. Broadly,

α has a larger effect on accuracy than does increasing K from 200 to 500; this is unsurprising, as a value of α very close to 1.0 renders temporality decreasingly relevant; so, the contribution of the top terms has less substantial effect on the final score than for lower values of α.

We observe that the accuracy on the WW dataset in Fig.9a is increasing with the increase of output size, K. The reason for this is as follows. In this evaluation, we evaluate the accuracy of the method using the approximate implementation, where we consider the recency and terms with different weights. Here we give more important/weight to the recent terms than the out-date ones. By doing so, it will make slight fluctuations in the results that some recent terms may have higher rank than the old terms. Therefore, frequent terms will interchange their rank slightly under our weighting schema, but overall in a very larger list of frequent terms (high value of K), the set of the most frequent terms is the same. This explains for the reason why with the higher value of K, it could make a higher accuracy as in the figure.

We also note that $\alpha = 1.0$ is equivalent to straightforward frequency counting (e.g., [1]) within the temporal window in which our dataset was collected. Fig. 9 therefore suggests that a time-decay model *is prerequisite* to effective TopK-TwTL approximation.



(a) On WW. **(b)** On CN.

Figure 10: Index size relative to the number of posts.

5.3. Index Construction and Maintenance

In these experiments, we evaluate several notions of "cost" for the tree data structures. As such, we differentiate between *QTi** and *QTb** as *incremental* and *batch* quad-tree construction, respectively. The R-tree only has an incremental (*RTi**) method, as batch construction of TwTL-enabled R-trees is quite inefficient: node splits that incur rebalance operations require rebuilding the lists.

5.3.1. Index size

Fig. 10 reports the size of the spatial index structure (quad-tree) and its time-weighted term lists relative to the number of tweets/weibos. As expected, the TopK-TwTL approximation requires significantly less memory than the exact Full-TwTL, ranging from 5.3–6.1×. Moreover, the index size grows linearly with the number of posts. In particular, considering that the raw sizes of the 15M, 30M, 60M, and 125M input WW datasets are 1.32, 2.64, 5.29, and 10.9 GB, respectively, the Full-TwTL requires similar space to the original data and the Top500-TwTL represents a 5–6-fold compression. With the CN dataset, the raw sizes of the input 10M, 20M, and 30.39M datasets are 1.3, 2.63, and 4.03 GB, respectively, the Full-TwTL also requires more space to the original data, and the Top500-TwTL represents a 2–5-fold compression on this dataset.

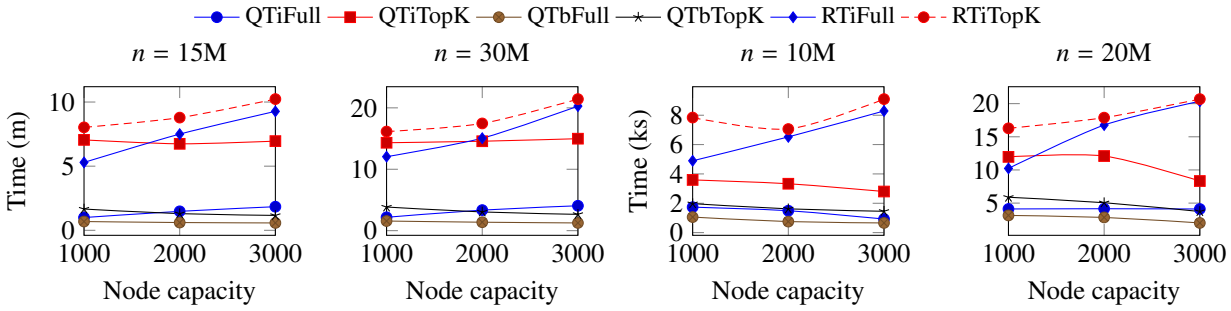
5.3.2. Construction time

Figs. 11–13 illustrate the time required (in thousands of seconds) to construct the complete spatial indexes with TwTL annotations (Algorithm 1), relative to node capacity, number of maintained counters (terms) K, and number of posts.

Node capacity. When building the index, if the number of posts stored in a leaf node reaches *MaxCapacity*, this node needs to be split and new child nodes are created. As splits are especially expensive, we first evaluate construction time as *MaxCapacity* varies from 1000–3000 posts (Fig. 11). Note that the *MaxCapacity* of RTi refers to the capacity of a leaf node, while the capacity of its internal nodes was set to 100 to reduce its restructuring cost.

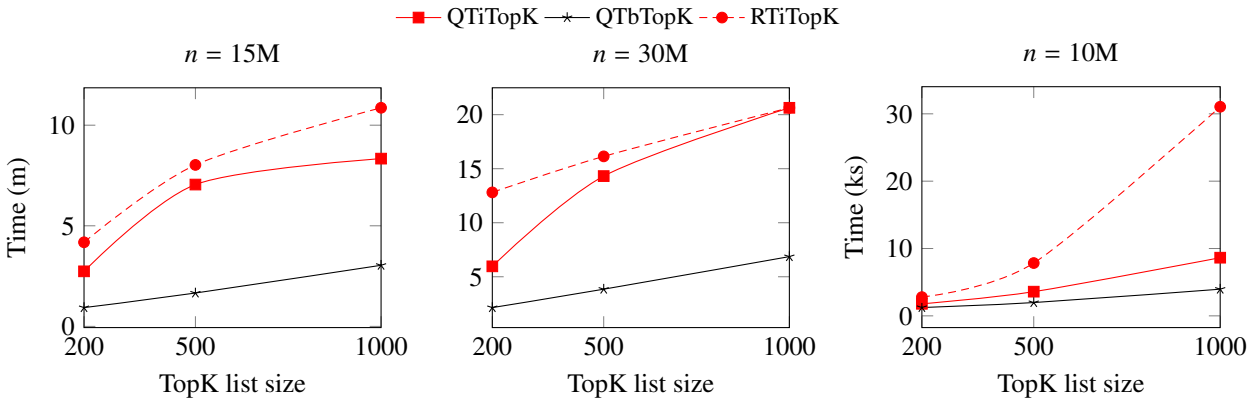
We observe that increases in *MaxCapacity* slow down the RTi* methods, but the quad-tree techniques are unaffected by this input parameter: this reflects the much higher cost of re-organising leaf node data when an R-tree node is split. We can also observe already an (expected) general trend: the cost of constructing an index over 30M tweets (right) is twice that of 15M tweets (left) for all methods. Although the TopK* approximation methods are faster in terms of query response time, they need more time to build its indices than the Full-TwTL methods. This is because of maintaining a TopK-TwTL in each node (Algorithm 2) is more expensive than updating a Full-TwTL, which simply merges all the weighted-frequencies for every term.

TopK list size. Fig. 12 shows construction time relative to the approximation threshold (the number of currently monitored terms) for the approximate methods. Note that the middle data point ($K = 500$) corresponds



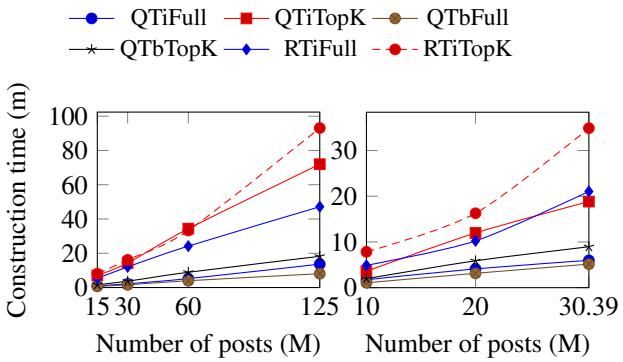
(a) On WW. **(b)** On CN.

Figure 11: Construction time relative to node capacity (m: minute, M: million).



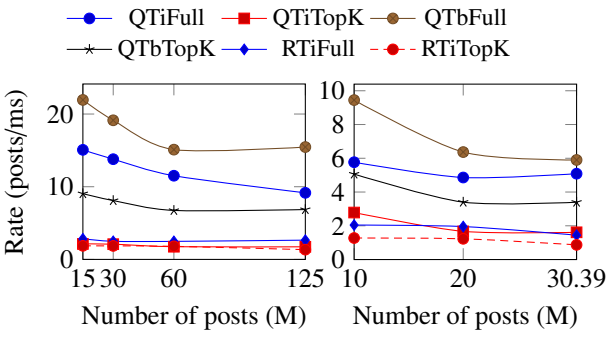
(a) On WW. **(b)** On CN.

Figure 12: Construction time relative to TopK list size (m: minute, M: million).



(a) On WW. **(b)** On CN.

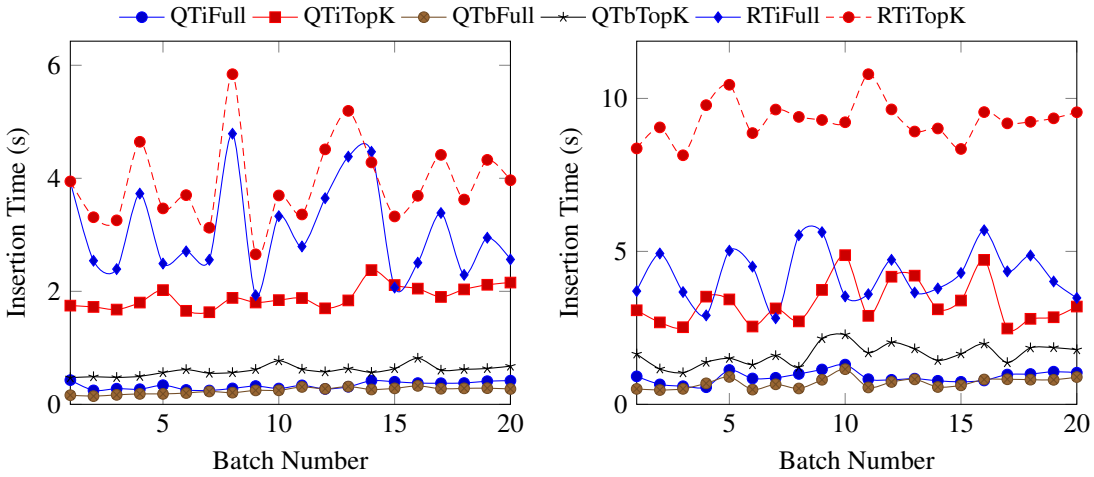
Figure 13: Construction time relative to number of posts (m: minute, M: million).



(a) On WW. **(b)** On CN.

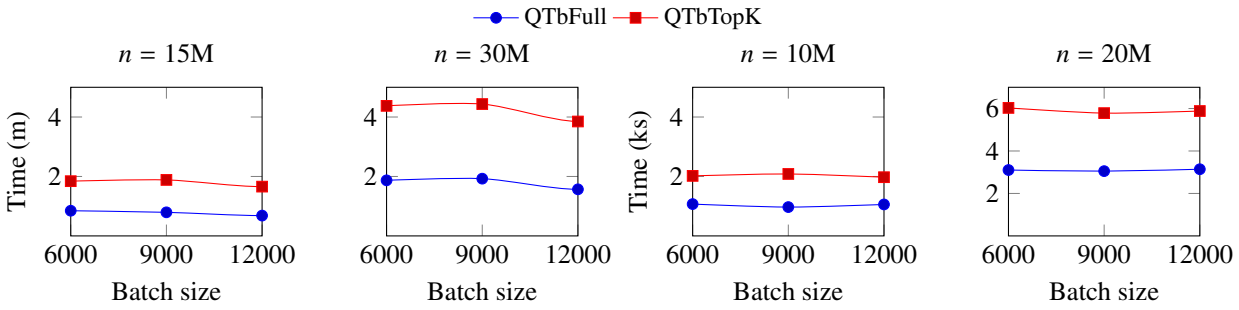
Figure 14: Digestion rate relative to number of posts.

exactly to the leftmost data point (capacity = 1000) in Fig. 11, and the result on CN 20M dataset is not showed because RTiTopK requires a large amount of time to finish when $K = 1000$. Here, the batch method (QTbTopK) scales linearly with K, whereas the incre-



(a) On WW. **(b)** On CN.

Figure 15: Insert time relative to batch number.



(a) On WW. **(b)** On CN.

Figure 16: Construction time relative to batch size (m: minute, M: million).

mental methods (QTiTopK and RTiTopK) are slower overall but scale sub-linearly. QTbTopK scales better, especially when the average text message is long (e.g., with the CN dataset), because of the buffering step when building its index, a map of arriving terms (B.M) was pre-summarized (lines 2–5, Algorithm 1), while the incremental methods insert post individually.

Post count. Fig. 13 shows construction time relative to the number of posts, which is an extension of the comparison already shown between the left and right plots of Figs. 11 and 12. Mostly, construction time matches the linear scaling of the index sizes observed in Fig. 10. The exception is RTiTopK, the degradation of which accelerates with n. We attribute this to the more expensive search cost of RTi*: an incremental insert first finds the correct insertion node before updating the logarithmic top-K priority queue. As n increases, so does this search cost, impacting the incremental construction time of the data structure. Moreover, as n increases, the number of (possibly upward propagating) node splits (and therefore TwTL reconstructions) also increases, as there are more nodes in the tree.

5.3.3. Index maintenance

The final experiments analyse the cost of batch maintaining the index in the presence of a stream of inserts.

Digestion rate. Fig. 14 shows the average number of posts (tweets/weibos) indexed per second, effectively the reciprocal of Fig. 13. QTb* obtains a stable digestion rate at $n \geq 60$M on WW dataset, and reaches a digestion rate 4.25× and 7.74× higher than QTi* and RTi*, respectively. On CN dataset, QTb* obtains a stable digestion rate at $n \geq 20$M, and reaches a digestion rate 1.82× and 4.62× higher than QTi* and RTi*, respectively.

Batch processing. Fig. 15 reports insertion time and its trend for inserting each consecutive batch of 12 000 tweets/weibos. The performance of RTi* is highly unstable, as the expensive node splits and TwTL recomputations occur randomly. By contrast, all four QT* methods insert each batch faster than either RTi* method and perform more stably. Fig. 16 shows the effect of batch size on QTb* construction: both methods improve slightly for batches of 12 000 tweets/weibos compared to 6 000 tweets/weibos.

As a summary, the extensive experiments and empirical analyses presented in this section have shown that the massive improvements in indexing and response times achieved by our methods make it possible to answer the real-time query kRFT and cope with the whole post stream.

6. Conclusion

In this work, we have introduced a new query type named *kRFT*, which returns the top-*k* locally popular terms in social stream data consisting of a huge amount of posts with high arrival rate. We propose both exact and approximate methods for answering this query efficiently. Our methods employ a quad-tree structure that admits batch insertion of posts to handle streaming workloads. Furthermore, a summary of time-weighted frequencies, TwTL, is maintained in the tree nodes and updated efficiently to support efficient query processing. The result set is computed by aggregating the summaries which belong to the query region. The exact method Full-TwTL maintains time-weighted counts of all terms, while the approximate method TopK-TwTL only maintains a relaxed TopK most time-weighted frequent terms. Extensive experiments are conducted to evaluate the proposed methods both on real-world and semi-synthetic datasets, and the results show that our framework is capable of working with real-life social post streams. In particular, the time cost for building index of the QTbTopK approach increases slowly when the dataset becomes larger. Further, it is able to return results rapidly with high accuracy.

References

[1] P. Ahmed, M. Hasan, A. Kashyap, V. Hristidis, V. J. Tsotras, Efficient computation of top-k frequent terms over spatio-temporal ranges, in: Proc. of the 2017 ACM SIGMOD, 2017, pp. 1227–1241.

[2] A. Skovsgaard, D. Sidlauskas, C. S. Jensen, Scalable top-k spatio-temporal term querying, in: the 30th IEEE ICDE, 2014, pp. 148–159.

[3] A. Magdy, A. M. Aly, M. F. Mokbel, S. Elnikety, Y. He, S. Nath, W. G. Aref, GeoTrend: spatial trending queries on real-time microblogs, in: Proc. of the 24th ACM SIGSPATIAL, 2016, pp. 7:1–7:10.

[4] Y. Xu, L. Chen, B. Yao, S. Shang, S. Zhu, K. Zheng, F. Li, Location-based top-k term querying over sliding window, in: Proc. of the 18th WISE, 2017, pp. 299–314.

[5] L. Chen, S. Shang, B. Yao, K. Zheng, Spatio-temporal top-k term search over sliding window, World Wide Web 22 (5) (2019) 1953–1970.

[6] L. Guo, D. Zhang, G. Li, K.-L. Tan, Z. Bao, Location-aware pub/sub system: When continuous moving queries meet dynamic event streams, in: Proc. of the ACM SIGMOD, 2015, pp. 843–857.

[7] L. Chen, S. Shang, Z. Zhang, X. Cao, C. S. Jensen, P. Kalnis, Location-aware top-k term publish/subscribe, in: Proc. of the 34th IEEE ICDE, 2018, pp. 749–760.

[8] A. Simitsis, A. Baid, Y. Sismanis, B. Reinwald, Multidimensional content eXploration, Proc. of the VLDB Endowment 1 (1) (2008) 660–671.

[9] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, in: Proc. of the 20th ACM SIGMOD-SIGACT-SIGART, 2001, pp. 102–113.

[10] R. Akbarinia, E. Pacitti, P. Valduriez, Best position algorithms for top-k queries, in: Proc. of the 33rd VLDB, 2007, pp. 495–506.

[11] L. Chen, G. Cong, C. S. Jensen, D. Wu, Spatial keyword query processing: An experimental evaluation, Proc. of the VLDB Endowment 6 (3) (2013) 217–228.

[12] A. Almaslukh, A. Magdy, Evaluating spatial-keyword queries on streaming data, in: Proc. of the 26th ACM SIGSPATIAL, 2018, pp. 209–218.

[13] M. Hadjieleftheriou, E. Hoel, V. J. Tsotras, Sail: A spatial index library for efficient application integration, GeoInformatica 9 (4) (2005) 367–389.

[14] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, K. Nørvåg, Efficient processing of top-k spatial keyword queries, in: Proc. of the 12th SSTD, 2011, pp. 205–222.

[15] Z. Li, K. C. Lee, B. Zheng, W.-C. Lee, D. Lee, X. Wang, IR-Tree: an efficient index for geographic document search, IEEE Transactions on Knowledge and Data Engineering 23 (4) (2011) 585–599.

[16] H.-J. Hong, G.-M. Chiu, W.-Y. Tsai, A single quadtree-based algorithm for top-k spatial keyword query, Pervasive and Mobile Computing 42 (2017) 93–107.

[17] S. Vaid, C. B. Jones, H. Joho, M. Sanderson, Spatio-textual indexing for geographical search on the web, in: Proc. of the 9th SSTD, 2005, pp. 218–235.

[18] A. Khodaei, C. Shahabi, C. Li, Hybrid indexing and seamless ranking of spatial and textual features of web documents, in: Proc. of the 21st DEXA, 2010, pp. 50–466.

[19] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, T. Suel, Text vs. space: Efficient geo-search query processing, in: Proc. of the 20th ACM CIKM, 2011, pp. 423–432.

[20] L. Chen, Y. Cui, G. Cong, X. Cao, Sops: A system for efficient processing of spatial-keyword publish/subscribe, Proc. VLDB Endow. 7 (13) (2014) 1601–1604.

[21] X. Wang, Y. Zhang, W. Zhang, X. Lin, W. Wang, AP-Tree: Efficiently support location-aware publish/subscribe, The VLDB Journal 24 (6) (2015) 823–848.

[22] A. Shraer, M. Gurevich, M. Fontoura, V. Josifovski, Top-k publish-subscribe for social annotation of news, Proc. VLDB Endow. 6 (6) (2013) 385–396.

[23] L. Chen, G. Cong, X. Cao, K. Tan, Temporal spatial-keyword top-k publish/subscribe, in: Proc. of the 31st IEEE ICDE, 2015, pp. 255–266.

[24] A. Metwally, D. Agrawal, A. E. Abbadi, An integrated efficient solution for computing frequent and top-k elements in data streams, ACM Transactions on Database Systems (TODS) 31 (3) (2006) 1095–1133.

[25] M. Dallachiesa, T. Palpanas, Identifying streaming frequent items in ad hoc time windows, Data & Knowledge Engineering 87 (2013) 66–90.

[26] G. Cormode, V. Shkapenyuk, D. Srivastava, B. Xu, Forward decay: A practical time decay model for streaming systems, in: 25th IEEE ICDE, 2009, pp. 138–149.

[27] Y. Lim, J. Choi, U. Kang, Fast, accurate, and space-efficient tracking of time-weighted frequent items from data streams, in: Proc. of the 23rd ACM CIKM, 2014, pp. 1109–1118.

[28] S. Wu, H. Lin, L. H. U, Y. Gao, D. Lu, Novel structures for counting frequent items in time decayed streams, World Wide Web 20 (5) (2017) 1111–1133.

[29] G. S. Manku, R. Motwani, Approximate frequency counts over data streams, in: Proc. of the 28th VLDB, 2002, pp. 346–357.

[30] G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, Journal of Algorithms 55 (1) (2005) 58–75.

[31] N. Homem, J. P. Carvalho, Finding top-k elements in data streams, Information Sciences 180 (24) (2010) 4958–4974.

[32] Y. Lim, U. Kang, Time-weighted counting for recently frequent pattern mining in data streams, Knowledge and Information Systems 53 (2) (2017) 391–422.

[33] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: Proc. of the ACM SIGMOD, 1984, pp. 47–57.

[34] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The r*-tree: An efficient and robust access method for points and rectangles, in: Proc. of the ACM SIGMOD, 1990, pp. 322–331.

[35] N. Beckmann, B. Seeger, A revised r*-tree in comparison with related index structures, in: Proc. of the ACM SIGMOD, 2009, pp. 799–812.

[36] T. Lee, B. Moon, S. Lee, Bulk insertion for r-trees by seeded clustering, Data & Knowledge Engineering 59 (1) (2006) 86–106.

[37] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, D. Šaulys, Trees or grids?: Indexing moving objects in main memory, in: Proc. of the 17th ACM SIGSPATIAL, 2009, pp. 236–245.

[38] I. Gargantini, An effective way to represent quadtrees, Communications of the ACM 25 (12) (1982) 905–910.

[39] G. R. Hjaltason, H. Samet, Speeding up construction of pmr quadtree-based spatial indexes, Proc. of the VLDB Endowment 11 (2) (2002) 109–137.

[40] G. M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, International Business Machines, 1966.

[41] P. Van Oosterom, T. Vijlbrief, The spatial location code, in: The 7th International Symposium on Spatial Data Handling, 1996, pp. 1–17.