Efficient Computation of Top-k Frequent Terms over Spatio-temporal Ranges

Pritom Ahmed
UC Riverside
pahme002@ucr.edu

Mahbub Hasan
UC Riverside
hasanm@cs.ucr.edu

Abhijith Kashyap
UC Riverside
akash001@cs.ucr.edu

Vagelis Hristidis
UC Riverside
vagelis@cs.ucr.edu

Vassilis J. Tsotras
UC Riverside
tsotras@cs.ucr.edu

ABSTRACT

The wide availability of tracking devices has drastically increased the role of geolocation in social networks, resulting in new commercial applications; for example, marketers can identify current trending topics within a region of interest and focus their products accordingly. In this paper we study a basic analytics query on geotagged data, namely: *given a spatiotemporal region, find the most frequent terms among the social posts in that region*. While there has been prior work on keyword search on spatial data (find the objects nearest to the query point that contain the query keywords), and on group keyword search on spatial data (retrieving groups of objects), our problem is different in that it returns keywords and aggregated frequencies as output, instead of having the keyword as input. Moreover, we differ from works addressing the streamed version of this query in that we operate on large, disk resident data and we provide exact answers. We propose an index structure and algorithms to efficiently answer such top-k spatiotemporal range queries, which we refer as *Top-k Frequent Spatiotemporal Terms (kFST)* queries. Our index structure employs an R-tree augmented by top-k sorted term lists (STLs), where a key challenge is to balance the size of the index to achieve faster execution and smaller space requirements. We theoretically study and experimentally validate the ideal length of the stored term lists, and perform detailed experiments to evaluate the performance of the proposed methods compared to baselines on real datasets.

Keywords

Top-K; Spatio-Temporal Databases; Social Networks

1. INTRODUCTION

Several online social media, such as Twitter, Instagram, Foursquare and Facebook, allow users to geotag their social posts. This creates novel data analytics problems, such as detecting popular topic trends or popular sites, most frequent trajectories, etc. In this paper, we investigate a basic query on geotagged social data. Given a user-specified spatiotemporal region, we want to find the k most frequent terms in the posts in this region. We refer to this problem as the *Top-k Frequent Spatiotemporal Terms (kFST)* Query. As an example, the user may want to know which terms have been popular around her current location over the past week, and thus her query specifies a spatial circle with radius 2 miles around her and a temporal interval of a week.

Our problem is different from recent works on the intersection of keyword search and spatial querying. These works generally return one or more spatial objects, given a query that specifies both a spatial and a keyword condition. In the context of geotagged social posts, these works would return one or more social posts. In contrast, our queries only specify a spatial condition and (the most frequent) terms are returned. Nevertheless, we leverage existing work on spatial and text indexing. There is also recent work addressing the streamed version of the kFST query in main memory; since we operate on large data that does not fit in memory, our focus is on creating efficient indexing while also providing exact answers. Section 2 discusses in detail previous related work.

A straightforward approach to address the kFST query, which we include as a baseline in our experiments, is to simply index all posts in an R-tree [16]. Given a query range the R-tree will provide all relevant posts in that range. To get the query answer, the terms of these posts need to be aggregated (using a group_by hash-based aggregation scheme to compute the frequency per term) and then sorted. Nevertheless, we found that performance degrades as the query region (and thus the posts involved) increases.

A first method we propose is to materialize a sorted term list (STL) for each leaf node of the R-tree, where each STL contains pairs of terms and their frequencies, sorted by decreasing frequency. To answer a kFST query we run a top-k algorithm [13] on the STLs involved in the query range. Note that the additional space used by the STLs on the leaf nodes does not increase the asymptotic space complexity of the index (it is still linear; moreover duplicate terms are aggregated). Unfortunately, performance will still deteriorate once there are too many STLs involved (i.e. the query region increases further), because top-k algorithms are known to degrade in performance when the number of lists is very large.

We thus extend the use of STLs to the inner nodes of the R-tree as well. Given a kFST query, if the MBR of an internal node is fully contained in the query region, the STL in that node is used in the top-k calculation (i.e. the search proceeds to lower nodes only when their MBRs are partially contained in the query region). This algorithm variant leads to fast query execution times, because the number of STLs accessed for the top-k calculation is significantly reduced. However, the inner level STLs add considerable

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA.

DOI: http://dx.doi.org/10.1145/3035918.3064032

extra space to our index, since an inner STL is aggregating over all the posts in its subtree (here each term is counted once for each level of the R-tree). This is especially critical in free text data objects, such as social posts, given the huge size of the vocabulary of user-generated content – typically in the millions of unique terms (including names, numbers, typos, and so on).

To maintain the advantage offered by inner STLs while keeping the indexing space low, we explore the use of *partial* STLs of *fixed* length λ. We create a model of the access patterns of the top-k algorithm on the indexing structure and a theoretical analysis, to determine the optimal length λ of the prefix of each STL that should be maintained, such as most top-k queries can be answered within the STL (if more entries are needed for a query, we invoke recursively a top-k algorithm on the child nodes). Note, that the use of partial STLs does not imply loss of accuracy; our algorithms return the exact top-k result.

In addition to the single-region variant of kFST, we present a multi-region variant, where the user may be interested in terms that are popular in one set of regions and not popular in another set of regions. For example, the user may be interested to know what is trending in her immediate neighborhood, but not trending in the whole city, which may indicate a local event.

The multi-region problem variant is challenging in terms of avoiding to access the same STLs multiple times – once for each query region that contains their MBR. Further, it is challenging to define a termination threshold of a top-k algorithm when the entries (terms) are sorted in the reverse order – in the above example, the terms in the whole city should be naturally sorted in increasing frequency because a lower frequency is more desirable, but the STL are always sorted by decreasing frequency. That is, the aggregation function is increasing on some STLs and decreasing on others.

The contributions of the paper can be summarized as follows:

- We propose STL-enhanced indexing and top-k algorithms to solve the kFST problem. Both Random Access (RA) and Non Random Access (NRA) variants are presented.
- We present a theoretical model to optimize the space requirements of the index structure by carefully pruning the length of the STL lists and experimentally evaluate it's accuracy.
- We experimentally explore the various indexing options from no STLs to full and/or partial STLs and identify the space versus query trade-offs.
- We extend our algorithms for the multi-region kFST problem and show that our multi-region approach is more efficient than simply running the single-region algorithm multiple times.

The rest of the paper is organized as follows: Section 2 discusses related work, while Section 3 formulates the problem. Section 4 describes our index structure and Section 5 presents the model used to estimate the STL size. We discuss an extension of our algorithms to accommodate multiple query regions in Section 6. Our indexing scheme and algorithms are evaluated in Section 7 while conclusions appear in Section 8.

2. RELATED WORK

Spatial Aggregation There has been work on spatial aggregation, where the goal is to efficiently compute the aggregate function (e.g. count or sum) of the main quantity of the application (e.g., sales) [21]. In our setting, this would solve the problem of computing the frequency of a specific keyword given a spatial area. However, kFST must handle millions of keywords and produce top-k frequency rankings.

Top-k Spatial Keyword Query : A top-k spatial keyword query retrieves the k objects that are closest to the query location and contain the query keywords [14, 11]. This problem has also been studied in the context of spatially-annotated web objects, where the goal is to combine both the textual content and the geolocation of Web pages when performing Web search [10, 12]. The work in [28] examines jointly processing multiple top-k spatial keyword queries, while the top-k spatial keyword query for continuously moving objects is studied in [29]. [20] examined spatiotemporal burstiness queries which, given a set of terms identify (unusual frequency) bursts of these terms in a given area. Those set of terms has to be provided beforehand and the system will process the stream of data to look for unusual spike in frequencies in those terms only.

More recently, the problem has been extended to return groups of spatial objects that satisfy some properties. Specifically, [9] solves the problem of retrieving a group of spatio-textual objects that collectively cover the query keywords, are close to the query location and have small inter-object distances; a generalization appears in [6]. Again, the problem is different than kFST as posts or users are returned and not terms. Several geo-social query variants are examined in [5]. One of them, closest to our work, is the problem of Top-k Frequent Social Keywords in Range, which computes the top-k terms based on their frequency in pairs of friends in a spatial range. Our problem differs because we aim to find the most frequent terms in a spatial range not restricted by any social media constraints. Even without this constraint, the algorithm in [5] will take too long because the inverted lists are stored only in the leaf nodes. In our experimental section we examine a similar approach that stores an inverted list in the leaf nodes only (termed as STL-L, Figure13). We find that as the size of the query region increases, this solution starts performing poorly compared to our proposed approaches.

Top-k Spatial Preference Queries: The work in [30] ranks objects based on their spatial neighborhood, i.e., find the top-k objects (e.g., homes) whose aggregate distance from other objects (e.g., restaurants) is minimized. A follow-up work solves a similar problem, except that there is a distance threshold, e.g., within 5 miles [25]. This problem is clearly different from kFST as we do not return posts but keywords of posts – we view each keyword in a post as a data point, and we find keywords with high density.

Top-k Spatio-Temporal Queries: The works in [22, 26] address the kFST query over streamed data on main memory; our work differs in that (i) we consider large datasets that reside on disk (thus indexing is necessary) and (ii) we provide exact (versus approximate) results. Geo-Trend [22] is a framework for computing top-k trending keywords over spatiotemporal ranges, i.e., terms whose frequencies are on the rise recently. A spatial grid is used while the time period depends on the size of the system's main memory. Only the top-k trending results in each spatiotemporal cell are maintained and combined to generate the query result.

AFIA [26] uses a multi-layer grid based index structure where each cell of the grid maintains the k+1 most frequent terms as materialized summary as opposed to our carefully pruned λ terms ($\lambda >> k$). 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, their 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. In addition to finding exact results, our work differs in that a model is introduced to identify the length of all materialized summaries.

Finally, GARNET [19] addresses various trending queries on mi-

croblogs; initially data is stored in main memory which is periodically flushed to disk. The framework can support multiple contexts including location. The spatial context is implemented by a fixed grid layer while the temporal domain uses a multilayer index. For each cell, and for each time unit (say day) they maintain a materialized top-k list. To answer a kFST query, for each cell included in the query region, they pick all lists that are included in the temporal query range and they run a top-k algorithm. Hence, it is not guaranteed that the exact top-k result can be computed (without having to access the raw tweets which defies the purpose of an index), as more than k entries may be needed from the cells in some queries. The time performance of their top-k algorithm (assuming it does not need to access the raw tweets) would be similar to the performance of our leaves-only, full-lists variant, which keeps an STL only at the leaf nodes of the index (note that our index combines the spatial and temporal dimensions). As we show in the experimental section, our other algorithms clearly outperform that approach.

3. PROBLEM DEFINITION

Let $\mathcal{D} = \{o_1, o_2, ..., o_N\}$ be a dataset with N objects, where each object $o \in \mathcal{D}$ corresponds to a post and consists of a pair of attributes $< Loc, Terms >$; $o.Loc$ is a 3-dimensional point that identifies the location of the post in space and time (e.g., $o.Loc$ is described by a triplet (x, y, T)). The attribute $o.Terms = \{t_1, t_2, ...\}$ denotes the collection of the post's terms (and may include duplicates). For simplicity in the following discussion (and examples) we consider only the spatial location of a post; this can be easily extended to add the post's timestamp T and thus support spatio-temporal queries (in Section 7 we also provide performance results under spatio-temporal ranges).

Let $V = \{\cup_{o \in \mathcal{D}} o.Terms\}$ be the vocabulary with all terms. Consider a dataset with 10 objects whose locations in 2D space are shown in Figure 1(a); the terms of these objects are shown in Figure 1(b). The vocabulary $\{\cup_{i=1}^{9} t_i\}$ contains 9 terms.

The frequency of a term $t \in V$ is denoted as $f(t) = \{f_{o_1}(t) + f_{o_2}(t) + ... + f_{o_N}(t)\}$, where $f_o(t)$ denotes the number of times t appears in $o.Terms$. Given a region R, the frequency of term t in R is denoted as $f_R(t) = \{\sum f_{o_i}(t) | o_i.Loc \in R\}$.

kFST Query Definition (*Single Region*): A kFST query Q is defined by the tuple $< R_Q, k >$, where R_Q denotes the region of interest and k denotes the number of output terms. The goal is to find the k terms : $t_1, t_2, ..., t_k$, whose frequencies $f_{R_Q}(t_1)$, $f_{R_Q}(t_2)$, ... , $f_{R_Q}(t_k)$ are the highest among all terms in V.

Consider the example in Figure 1. The dotted region in Figure 1(a) denotes the query region R_Q. Assume the user is interested in the top-2 terms ($k = 2$). Therefore, the goal is to compute the two terms from $\{\cup_{i=1}^{9} t_i\}$ whose frequencies are the maximum in the dotted region (i.e. in the $Terms$ of five objects $\{o_1, o_2, o_3, o_6, o_7\}$).

We also explore the *multi-region extension* of kFST: $< S^+, [S^-], k >$, which provides two sets of regions S^+ (for *inclusion*) and an optional S^- (for *exclusion*) and identifies the top-k terms that are popular in the S^+ regions and not popular in the S^- regions. Terms in S^+ are penalized if they are popular in any of the S^- regions. If only S^+ is provided, kFST simply combines multiple regions and identifies the top-k terms (if regions in S^+ overlap, common posts are not duplicated). As an example consider finding the top-k most frequent terms in posts from all the Ivy League campuses over 2015 (S^+). Based on the application, the user may choose to normalize the term frequencies per campus. We can also identify the terms which were most discussed in the Ivy League campuses and were not popular in the US campuses over the same period (S^-). Note that this inclusion/exclusion can extend to higher dimensional regions by adding more attributes to the R-tree.

4. PROPOSED INDEX STRUCTURE AND ALGORITHMS

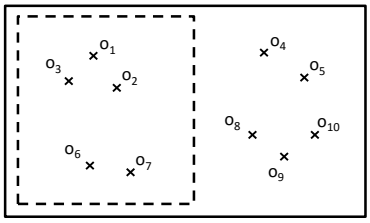
We assume that the full set $\mathcal{D}$ of posts is large and stored on disk. Figure 2(a) shows the *baseline* approach that indexes the posts with a multidimensional R-tree [16]. We use R-tree and not other temporal indexes for spatiotemporal data, because we are indexing points and not intervals. To solve the $< R_Q, k >$ query, we access only the posts contained in R_Q; their terms are collected, ordered and the top-k terms are returned. Next, we present a suite of indexes that enhance the R-tree with STLs and corresponding top-k algorithms to efficiently solve the kFST problem, without having to scan all objects that match the query region. Our approaches differ on which tree nodes (leaf/index) contain STLs and on whether these STLs are full or partial. In the following discussion we refer to each approach using the notation in Table 1.

4.1 Full Lists on Leaf Nodes Only (STL-L)

Since the number of terms can rapidly increase with R_Q, a better approach to compute the top-k terms, is to store sorted term lists (STLs) for the leaf nodes of the R-tree. In particular, the STL of a leaf node n_l contains the aggregated term entries from the object (posts) stored within the node's MBR R_{n_l}, sorted based on the frequencies of the terms in that MBR. The total number of entries in this STL, i.e. the vocabulary size, is $|V_{n_l}|$ where $V_{n_l} = \{\cup_{o \in \mathcal{D}} o.Terms | o.Loc \in R_{n_l}\}$. For each term $t \in V_{n_l}$, we have a term entry of the form $< t, t.ObjectEntries, t.Freq >$, where $t.ObjectEntries$ is a list of object entries that contain t. Each entry in this list has the form $< Loc, Freq > \equiv [\exists o \in \mathcal{D} : Loc = o.Loc \in R_{n_l} \; and \; Freq = f_o(t) > 0]$. Finally, the third field $t.Freq$ is the sum of all $Freq$ values of the object entries in $t.ObjectEntries$. The term entries in STL are sorted by their $Freq$ values in descending order. Figure 2(b) shows the STLs for the two leaf nodes R_3 and R_4 of the R-tree in Figure 2(a). We refer to this indexing scheme as *STL-L*, to denote that only the leaf nodes of the R-tree have STL lists.

Algorithm Overview: To leverage the STL-L index we proceed in two steps. First, the leaf nodes that intersect with the query region R_Q are identified; then a top-k algorithm is applied on the STLs of the intersected leafs nodes. If a leaf node is fully contained in R_Q its STL is used directly in the top-k algorithm; if it is partially contained, then a partial STL list is created with the objects that are contained in R_Q, as explained below. Among the several top-k algorithms in literature, we use two popular variations, the Random Access (RA) and Non Random Access (NRA) [13, 24]. Note that RA is a modified (improved) version of TA. The specific improvements are presented in Section 4.4.

Random Access (RA): At each iteration i, the RA algorithm extracts the i^{th} term entry t_e from each of the involved leaf STLs. The sum of all $t_e.Freq$ values is considered as the threshold θ at iteration i. Each time a new term t is seen, RA scans the other STLs to compute the aggregate $f_{R_Q}(t)$. Note that in our case, for a given term entry t_e with the term t, if the leaf node n_l of the STL (that contains t_e) is fully contained in R_Q then $t_e.Freq$ is used in the $f_{R_Q}(t)$ computation. Otherwise, $t_e.ObjectEntries$ is scanned to compute $f_{R_Q \cap R_{n_l}}(t)$ which contributes to the $f_{R_Q}(t)$. RA stops when it finds k terms whose frequencies are higher or equal to the threshold θ value. As an example, consider the dotted query region in Figure 1(a). Based on the R-tree in Figure 2(a), the

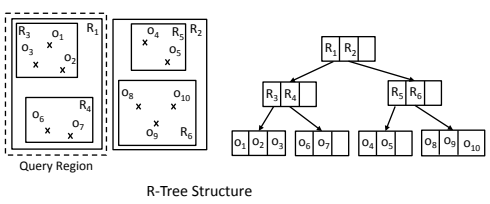


(a) Locations

Object	Terms	Object	Terms
o_1	$\{t_1, t_2, t_4, t_6\}$	o_6	$\{t_1, t_2, t_5, t_9\}$
o_2	$\{t_2, t_2, t_4\}$	o_7	$\{t_1, t_1, t_4\}$
o_3	$\{t_1, t_3, t_4\}$	o_8	$\{t_4, t_5, t_6, t_9\}$
o_4	$\{t_6, t_7, t_8, t_8\}$	o_9	$\{t_2, t_4, t_9\}$
o_5	$\{t_4, t_5, t_9\}$	o_{10}	$\{t_2, t_6, t_7\}$

(b) Terms

Figure 1: Sample dataset containing 10 objects, (a) shows the locations and (b) shows the terms of the objects.



(a)

Term	ObjectEntries	Freq
t_2	<o_1.Loc,1>, <o_2.Loc,2>	3
t_4	<o_1.Loc,1>, <o_2.Loc,1>, <o_3.Loc,1>	3
t_1	<o_1.Loc,1>, <o_3.Loc,1>	2
t_3	<o_3.Loc,1>	1
t_6	<o_1.Loc,1>	1

STL of R_3

Term	ObjectEntries	Freq
t_1	<o_6.Loc,1>, <o_7.Loc,2>	3
t_2	<o_6.Loc,1>	1
t_4	<o_7.Loc,1>	1
t_5	<o_6.Loc,1>	1
t_9	<o_6.Loc,1>	1

STL of R_4

(b)

Figure 2: Spatial R-Tree for the sample dataset in Figure 1 and leaf level STLs

Index	Description
STL-L	full lists on leaf nodes only
STL-LI	full lists on leaf and index nodes
STL-Li	full lists on leaf nodes, partial lists in index nodes
STL-li	partial lists on leaf and index nodes

Table 1: Different index types.

two leaf nodes R_3 and R_4 are fully contained in the query region. Therefore RA executes on these two nodes' STLs (Figure 2(b)).

Non Random Access (NRA): Similarly, NRA scans all the STLs involved in top-k computation in parallel. Each time a new term t is seen, NRA computes a Best Score and a Worst Score for that term (for details see [13]) and stops when it finds k terms whose Worst Scores are higher or equal to the threshold θ value.

4.2 Full Lists on All Nodes (STL-LI)

Solving the kFST query using leaf level STLs shows good performance for relatively small query regions. However, as the size of R_Q increases, the number of intersected leaf nodes and thus the number of involved STLs increases. This slows down both the RA and NRA performance (see Figures 13a, 13b in the experimental evaluation). One solution is to enhance our index structure adding STLs to all inner level nodes of the R-tree. Figure 3 shows the STLs for inner level nodes R_1 and R_2. Note that the $ObjectEntries$ fields are removed from the term entries of inner level STLs. This is to improve space efficiency; we refer to this scheme as *STL-LI*.

Term	Freq
t_1	5
t_2	4
t_4	4
t_3	1
t_5	1
t_6	1
t_9	1

STL of R_1

Term	Freq
t_4	3
t_6	3
t_9	3
t_2	2
t_7	2
t_8	2
t_5	2

STL of R_2

...

Figure 3: Inner level STLs.

Using the additional inner level STLs, we consider a modified tree traversal algorithm (Algorithm 2). Starting from the root node of the R-tree, if an inner level node is fully contained in the query region, then no further checking is required for the children of that node. The STL of this fully contained node is used in the top-k computation. However, if an inner level node overlaps with the query region then its children nodes are checked. This process continues until we reach the leaf level where the leaf nodes that intersect with the query region are identified. As before, if a leaf node is not totally contained in the query region then we create an STL only for the node's posts that are contained.

Using the modified tree traversal algorithm, consider again the query region in Figure 1(a). Based on the R-tree in Figure 2(a), only the inner level node R_1 is fully contained in the query region. Therefore, no further checking is done and the top-k terms are returned by the top-K algorithms (both RA and NRA) using only the STL of R_1. This reduces the number of involved STLs from 2 to 1.

4.3 Considering Partial Lists(STL-Li, STL-li)

Unfortunately the STL-LI approach requires large space especially for the STLs at the higher level nodes. Figure 4 shows the average number of term entries of a STL at different levels (level 4 corresponds to the index root). The number of term entries (and thus the size of a STL) increases for the higher levels. Nevertheless, we note that, both RA and NRA typically scan only a *small subset* of the entries in a STL. We can thus exploit this early stopping property to compute the **expected number of accessed term entries** (λ) to be stored in an STL (addressed in Section 5). Using this λ value we shrink the size of each inner level STL, which reduces the overall space. The leaf level STLs still keep their full size (in case the threshold algorithms cannot stop within the λ-sized inner STLs, the leaf STLs can then provide any additional terms needed). We refer to this index as *STL-Li*.

We proceed with how the threshold algorithms need to be modified for the kFST problem given that the inner level STLs contain λ number of term entries while the leaf level STLs contain all term entries. The overall approach is depicted in Algorithm 1. At first, the algorithm finds the R-tree nodes whose STLs are involved in top-k computation (line 1). After that, it executes the RA or NRA to compute the top-k term entries using Algorithm 3 or 4 re-

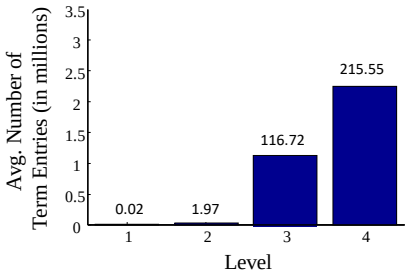


Figure 4: Level vs. avg. number of term entries per full STL. The number on top of each bar is the avg. size of that level STL in MB.

Algorithm 1 $kFST-STL(R_Q, k, root)$

Require: Query Region R_Q, number of output terms k and the root node of R-tree $root$
Ensure: Return top-k terms with highest frequencies in R_Q
1: $\mathcal{N} \leftarrow FindCandidateNodes(R_Q, root)$
2: $\mathcal{E} \leftarrow$ RA-$STL(\mathcal{N}, R_Q, k)$ / $NRA\text{-}STL(\mathcal{N}, R_Q, k)$
3: $\mathcal{T} \leftarrow \emptyset$
4: **for** each term entry t_e in $\mathcal{E}$ **do**
5: $\quad \mathcal{T} \leftarrow \mathcal{T} \cup t_e.Term$
6: **return** $\mathcal{T}$

spectively (line 2). The main component of these algorithms is the repeat-until loop (lines 2 - 24 and lines 2 - 22 respectively).

The RA version (Algorithm 3), at each iteration i, scans the i^{th} term entry from each of the involved STLs (lines 5 - 8) and computes the θ value (line 9). If the index i exceeds λ, Algorithm 5 is called (line 8) for each involved inner level STL to compute the next term entry in sorted order. Algorithm 5 invokes Algorithm 3 recursively for the STLs of the child nodes. If a new term t is seen, Algorithm 3 looks up the term t in all involved STLs and computes the aggregate frequency (lines 11 - 20). Algorithm 6 is a supporting method used to compute the value of a term entry t_e in R_Q. It scans all the object entries in $t_e.ObjectEntries$, finds the object entries that are contained in R_Q and aggregates their frequencies.

The NRA version (Algorithm 4), at each iteration i, scans the i^{th} term entry from each of the involved STLs (lines 5 - 8) and computes the $tops$ value (line 9) which is the maximum possible value for any of the unseen term that may appear in any of the later scan in that particular STL. Note that, when the index i exceeds λ, Algorithm 5 is called (line 8) for each involved inner level STL to compute the next term entry in sorted order. Algorithm 5 invokes Algorithm 4 recursively for the STLs of the child nodes. If a new term t is seen, Algorithm 4 adds it to the buffer called $SortedTopKElements$. Then the Algorithm updates the partial score of the term in the $SortedTopKElements$. This partial score is used to calculate the bestScore and worstScore of each of the terms in the buffer $SortedTopKElements$. The threshold value θ is calculated in line 21 which is the summation of $tops$ for all the STLs involved in the calculation.

The advantage of using full STLs at the leaf nodes comes at the expense of the full list space overhead. A remaining question is whether we can actually reduce this overhead further. Such an approach would replace each full leaf STL with a partial one. In the experimental section we term this approach as *STL-li*. We note that a leaf node has a limited number of objects (based on the fixed page size); thus if needed we could still compute the full STL.

Clearly, λ depends on k (see Section 5). Hence, k needs to be known when building the partial lists. We argue that this is a reasonable assumption for several applications. For example, Twitter as of now displays the top-10 trending topics (or hashtags) for each user (our work would allow for a more fine grained list of topics per user). Other applications displayed on mobile screens have similar constraints for k. Further, in the experimental section (Figure 15) we show that the proposed algorithm performs well for up to 50% larger k than the one originally provided.

Algorithm 2 $FindCandidateNodes(R_Q, n)$

Require: Query Region R_Q, and the node n
Ensure: Return the set of candidate nodes from the subtree rooted at n such that the STLs of the selected nodes are used for top-k computation
1: **if** $IsLeaf(n)$ **then**
2: $\quad$ **if** $R_Q \cap R_n \neq \emptyset$ **then**
3: $\quad\quad$ **return** $\{n\}$
4: $\quad$ **else**
5: $\quad\quad$ **return** $\emptyset$
6: **if** $R_n \subseteq R_Q$ **then**
7: $\quad$ **return** $\{n\}$
8: **else**
9: $\quad \mathcal{N} \leftarrow \emptyset$
10: $\quad$ **for** each child c of n **do**
11: $\quad\quad \mathcal{N} \leftarrow \mathcal{N} \cup FindCandidateNodes(R_Q, c)$
12: $\quad$ **return** $\mathcal{N}$

Replace line 9 by: 9(a): $\theta \leftarrow \theta + t_e.Freq$ 9(b): $tops[n] \leftarrow t_e.Freq$
Replace line 10 by: 10(a): $\mathcal{T} \leftarrow t_e.Term$ 10(b): **for** each t in $\mathcal{T}$
Add the following lines between line 20 and 21: (i) $index \leftarrow$ index of n' in $\mathcal{N}$ (ii) $maxPossible \leftarrow f' + \sum_{i \leftarrow index}^{i<=Size(\mathcal{N})} tops[i]$ (iii) **if** Size($\mathcal{E}$) > k **and** maxPossible < $\mathcal{E}$[k].Freq **break**

Table 2: Changes on RA-STL to reduce random accesses.

4.4 Optimizations to the top-k Algorithms

Optimizing RA-STL: A standard RA algorithm makes random accesses for all the terms it has seen. However, after the buffer has at least k elements, there exist some terms which can never make it to the top-k. As a result, we can avoid making random accesses for such terms. Table 2 shows the necessary changes for this optimization. We are using the array $tops$ to keep track of the values of each list/node in the current iteration. Later, we use this value to calculate the maximum possible value ($maxPossible$) for each term we encounter. If the k-th term in the buffer already has a greater value that this maximum possible value, no further random accesses are needed for that keyword.

We experimentally evaluated this optimization, using the setting described in Section 7.1. The results appear in Figure 5 using the RA-STL-Li as an example. The optimization speeds up the query performance by around 7 times on average. In the rest of the paper, all RA-STL algorithms use the optimized approach.

Optimizing NRA-STL: After implementing the NRA-STL algorithm, we observed that the CPU time required for sorting the

Algorithm 3 $RA\text{-}STL(\mathcal{N}, R_Q, k)$

Require: Set of nodes $\mathcal{N}$, the query region R_Q and the number of output term entries k
Ensure: Execute Random Access TA on the STLs of the nodes in $\mathcal{N}$ and return the top-k term entries with highest frequencies

$\mathcal{E} \leftarrow \emptyset$, $i \leftarrow 0$
repeat
  $\theta \leftarrow 0$, $f \leftarrow 0$
  for each node n in $\mathcal{N}$ **do**
    $L \leftarrow getSTL(n)$
    $t_e \leftarrow i^{th}$ term entry in L
    if t_e is null **and** $NotIsLeaf(n)$ **then**
      $t_e \leftarrow GetTermEntry(n, R_Q, i)$
    $\theta \leftarrow \theta + t_e.Freq$
    $t \leftarrow t_e.Term$
    if t has not been seen yet **then**
      $f' \leftarrow 0$
      if $isLeaf(n)$ **then**
        $f' \leftarrow ComputeTermFreq(t_e, R_Q)$
      else
        $f' \leftarrow t_e.Freq$
      for each node n' in $\mathcal{N}$ **do**
        if $n' \neq n$ **then**
          do random access for term t on STL of n' and compute $f_{R_Q \cap R'_n}(t)$
          $f' \leftarrow f' + f_{R_Q \cap R'_n}(t)$
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{< t, \emptyset, f' >\}$
  $f \leftarrow$frequency of the k^{th} term entry in $\mathcal{E}$
  $i \leftarrow i + 1$
until $\theta > f$
return top-k term entries in $\mathcal{E}$ with highest frequencies

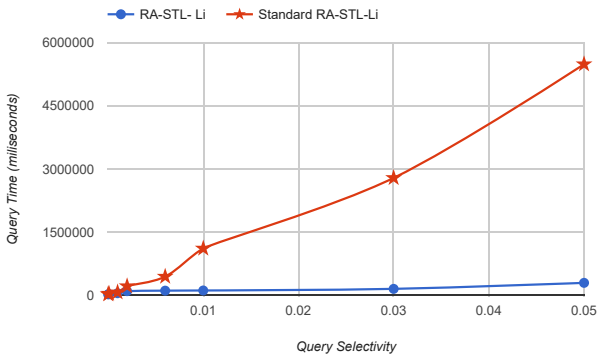


Figure 5: Comparison between standard RA-STL-Li and our optimized RA-STL-Li.

$SortedTopKElements$ is high. We thus applied the following optimizations. (i) **Reduce Buffer Size :** The work in [23] showed that if the summation of the last seen value in all lists is less than the k-th highest score in buffer then no keyword which has not been seen in any input can end up in the top-k result. Hence once that condition is true we do not add new keywords in the buffer. (ii) **Use QuickSelect:** Initially, we were sorting the whole buffer to get the k-th worst value (needed to decide if the calculation of top-k is finished). Instead we use the QuickSelect algorithm [17] to fetch the k-th highest value from the buffer *without* sorting it. (iii) **Delay Checking for Finish:** We check if the algorithm is finished in every 50 steps rather than every step. This significantly reduces the

Algorithm 4 $NRA\text{-}STL(\mathcal{N}, R_Q, k)$

Require: Set of nodes $\mathcal{N}$, the query region R_Q and the number of output term entries k
Ensure: Execute NRA on the STLs of the nodes in $\mathcal{N}$ and return the top-k term entries with highest frequencies

$SortedTopKElements \leftarrow \emptyset$, $i \leftarrow 0$
repeat
  $\theta \leftarrow 0, f \leftarrow 0$
  for each node n in $\mathcal{N}$ **do**
    $L \leftarrow getSTL(n)$
    $t_e \leftarrow i^{th}$ term entry in L
    if t_e is null **and** $NotIsLeaf(n)$ **then**
      $t_e \leftarrow GetTermEntry(n, R_Q, i)$
    $tops[n] \leftarrow t_e.Freq$
    $t \leftarrow t_e.Term$
    if t has not been seen yet **then**
      $SortedTopKElements \leftarrow tke(t)$
    else
      $tke \leftarrow SortedTopKElements(t)$
    if $isLeaf(n)$ **then**
      $tke.partialScore \leftarrow tke.partialScore + ComputeTermFreq(t_e, R_Q)$
    else
      $tke.partialScore \leftarrow tke.partialScore + t_e.Freq$
    $SortedTopKElements \leftarrow tke(t)$
  $\theta \leftarrow \sum_n tops[n]$
  $i \leftarrow i + 1$
until $SortedTopKElements[k].worstScore \geq \theta$
return top-k term entries in $\mathcal{E}$ with highest frequencies

Algorithm 5 $GetTermEntry(n, R_Q, i)$

Require: The node n, the query region R_Q and the index i
Ensure: Return the i^{th} term entry in the region R_n

$\mathcal{N} \leftarrow \emptyset$
for each child c of n **do**
  $\mathcal{N} \leftarrow \mathcal{N} \cup c$
$\mathcal{E} \leftarrow (N)RA - STL(\mathcal{N}, R_Q, i)$
return the i^{th} term entry in $\mathcal{E}$

CPU time. (iv) **Sorting Efficiently:** In case of ties among the term scores in the buffer, the standard NRA algorithm sorts them based on the best possible score. However, calculating the best possible score involved redundant computations. Instead we are only sorting the keywords whose worst possible score is equal to the k-th worst possible score in the buffer.

Figure 6 depicts the experimental evaluation of these optimizations using NRA-STL-Li as example. The improvement over the standard NRA-STL-Li is drastic (on average 50 times). In the rest, all NRA-STL algorithms use the optimized approach.

5. COMPUTING THE EXPECTED STL SIZE

We would like to estimate how long the ranked lists of internal nodes should be so as to minimize the chance that the top-k algorithms of Section 4.3 will need to access more terms, while at the same time keeping the length of the lists short to save space. For that, we estimate the expected STL size λ accessed by our top-k algorithms in two steps. In the first step, we estimate vector $M = (m_1, m_2, ..., m_h)$, where m_i denotes the expected number of STLs involved in the top-k calculation from level i; h is the height of the R-tree. Using M, we calculate λ in the second step.

Algorithm 6 $ComputeTermFreq(t_e, R_Q)$

Require: The term entry t_e, and the query region R_Q
Ensure: Return the frequency of t_e in the region R_Q
1: $f \leftarrow 0$
2: **for** each object entry o_e in $t_e.ObjectEntries$ **do**
3: **if** $o_e.loc \in R_Q$ **then**
4: $f \leftarrow f + o_e.freq$
5: **return** f

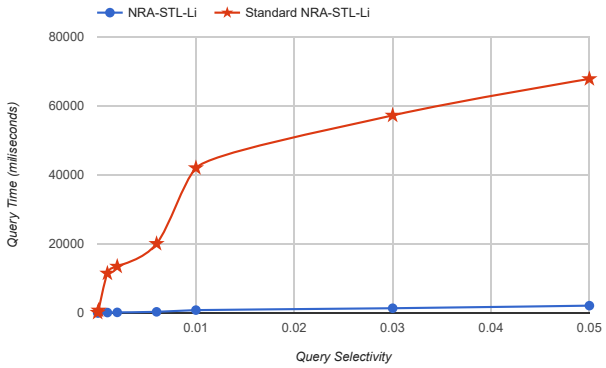


Figure 6: Comparison between standard NRA-STL-Li and our optimized NRA-STL-Li.

Step 1 - Calculate *M*: Given the query region R_Q, we start from the root level (level h) of the R-tree. At each inner level i ($2 \leq i \leq h$), we estimate the expected number of nodes which are fully contained in the query region and the region covered by the contained nodes. The STLs of the contained nodes are involved in the top-k calculation. Thus, m_i for inner levels is equal to the number of contained nodes. The remaining query region (i.e. the query region which is not covered by the contained nodes) is used as the new query region for the next level $i-1$. This process continues until we reach the leaf level ($i = 1$) where the number of nodes that intersect with the new query region is estimated as m_1.

In the discussion below we use the parameters in Table 3. Consider a d-dimensional unit dataspace ($[0,1)^d$) which contains the N objects. An R-tree with height h and average node capacity (fanout) f stores these N objects. Let, N_i be the number of nodes at level i and $S_i = (s_{i,1}, s_{i,2}, ..., s_{i,d})$ be the average size of a level i node. Given N and f, to estimate the R-tree properties (h,N_i,S_i) we use the analysis described in [27].

Since N objects are contained in N_1 nodes at leaf level and the average fanout factor is f, the number of leaf level nodes is $N_1 = \frac{N}{f}$. Similarly N_1 nodes are contained in N_2 nodes at level 2, therefore $N_2 = \frac{N}{f^2}$. Thus the number of nodes at level i is,

$$N_i = \frac{N}{f^i} \tag{1}$$

The height h of the R-tree is calculated as [27, 15],

$$h = 1 + \lceil \log_f \frac{N}{f} \rceil \tag{2}$$

To compute S_i, we assume that the node sides are equal in all dimensions (i.e. $s_{i,1} = s_{i,2} = ... = s_{i,d}$). Let s_i be the average size of a level i node in all dimensions. Since f number of level $(i-1)$ nodes are contained in a single node at level i, the number of level $(i-1)$ nodes that contribute to a single side of level i node

Symbol	Description
$M = (m_1, m_2, ..., m_h)$	expected number of STLs involved in the top-k calculation from different level
h	height of R-tree
$S_i = (s_{i,1}, s_{i,2}, ..., s_{i,d})$	avg. size of an MBR at level i
$N_r = (N_1, N_2, ..., N_h)$	number of MBRs at different levels
$R_Q = (q_1, q_2, ..., q_d)$	size of query region
f	avg. node capacity (fanout)
N	number of objects

Table 3: Model Parameters

is $\sqrt[d]{f}$. Therefore s_i can be computed as,

$$s_i = (f^{1/d} - 1).\frac{1}{(N_{i-1})^{1/d}} + s_{i-1} \tag{3}$$

Here $(N_{i-1})^{1/d}$ is the average distance between the centers of two consecutive level $(i-1)$ node projections in a single dimension. The detailed analysis is described in [27].

For simplicity, we assume that the query sides of R_Q are also equal in all dimensions (i.e. $q_1 = q_2 = ... = q_d = q$). Given N_i and s_i, the number of level i nodes that intersect with query region q^d is [27],

$$intersect(N_i, s_i, q) = N_i.(s_i + q)^d \tag{4}$$

As stated earlier, here we are interested in the estimation of the number of fully contained nodes for inner levels (which is a subset of intersected nodes computed in [27]). The next analysis describes how we can estimate the number of contained nodes given the values N_i, s_i and q.

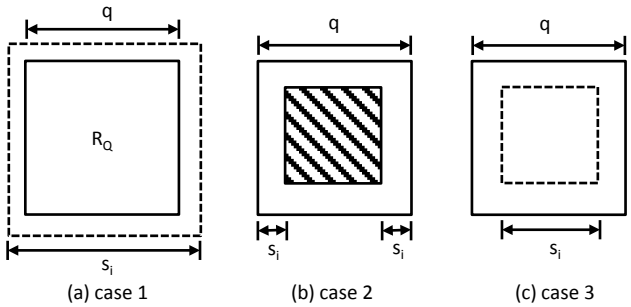


Figure 7: Case analysis of s_i and q

Based on the values of s_i and q, there are three possible cases,

Case 1 ($s_i > q$)**:** The node size is greater than the query region (Figure 7(a)). Therefore, no node is contained in the query; this means we have to go to next level $(i-1)$.

Case 2 ($q \geq 2s_i$)**:** In this case, we consider a d-dimensional rectangle (the shaded region in Figure 7(b)) inside the query region where each side of the inner rectangle is s_i distance far away from the query rectangle. We argue that the nodes which intersect with the inner rectangle are the nodes that are contained in the query region. The number of nodes that intersect with the inner rectangle is $intersect(N_i, s_i, q - 2s_i)$. Let A_i be the region covered by the contained nodes. Using a similar analysis as for the s_i calculation, we can estimate the average size a_i of a single side of A_i, by,

$$a_i = (intersect(N_i, s_i, q - 2s_i)^{1/d} - 1).\frac{1}{(N_i)^{1/d}} + s_i \tag{5}$$

The remaining uncovered region (i.e. $q^d - (a_i)^d$) is considered as the new query region for the next level which can be divided into small rectangles of size $(\frac{q-a_i}{2})^d$. The number of small rectangles is estimated as $\lceil (q^d - (a_i)^d)/(\frac{q-a_i}{2})^d \rceil$

Case 3 ($s_i \leq q < 2s_i$)**:** In this case, only one node can be contained in the query region assuming no overlapping between the nodes at a given level (Figure 7(c)). This assumption is a reasonable property for a good R-tree [7]. Using the similar analysis explained in case 2, the remaining uncovered region (i.e. $q^d - (s_i)^d$) is divided into $\lceil (q^d - (s_i)^d)/(\frac{q-s_i}{2})^d \rceil$ small rectangles of size $(\frac{q-s_i}{2})^d$. These small rectangles are considered as the new query region for the next level.

To compute m_1, we first compute the total number of leaf nodes covered by the contained inner level nodes. We compute this value as $\sum_{i=2}^{h} m_i f^{i-1}$ since each contained node at level i ($2 \leq i \leq h$) covers total f^{i-1} number of leaf nodes. We then subtract this value from the total number of leaf nodes that intersect with the original query q^d. The exact formula used for m_1 computation is,

$$m_1 = intersect(N_1, s_1, q) - \sum_{i=2}^{h} m_i f^{i-1} \quad (6)$$

Algorithm 7 shows the pseudocode to compute M. At first, lines $1-4$ compute the R-tree properties h, N_i and s_i. Then using the R-tree properties, the second for loop (lines $7-21$) computes the M. Each iteration of the for loop corresponds to a level of R-tree. Lines $8-9$ compute m_1 for the leaf level and lines $10-21$ compute m_i for the inner levels ($2 \leq i \leq h$). The variable $factor$ stores the number of query rectangles at a level i.

Algorithm 7 $ComputeM(N, f, d, q)$

Require: The number of objects N, the fanout factor f, the dimensionality d and the average size of query region side q
Ensure: Return the vector M
1: Calculate h using Equation 2
2: $s_0 \leftarrow 0$
3: **for** $i \leftarrow 1$ to h **do**
4: Calculate N_i and s_i using Equations 1 and 3 respectively
5: $factor \leftarrow 1$
6: $q' \leftarrow q$
7: **for** $i \leftarrow h$ to 1 **do**
8: **if** $i = 1$ **then**
9: Calculate m_1 using Equation 6
10: **else**
11: **if** $q' < s_i$ **then**
12: $m_i \leftarrow 0$
13: **else if** $q' \geq 2s_i$ **then**
14: $m_i = factor \times intersect(N_i, s_i, q' - 2s_i)$
15: Calculate a_i using Equation 5
16: $factor \leftarrow factor \times \lceil ((q')^d - (a_i)^d)/(\frac{q'-a_i}{2})^d \rceil$
17: $q' \leftarrow \frac{q'-a_i}{2}$
18: **else**
19: $m_i \leftarrow factor \times 1$
20: $factor \leftarrow factor \times \lceil ((q')^d - (s_i)^d)/(\frac{q'-s_i}{2})^d \rceil$
21: $q' \leftarrow \frac{q'-s_i}{2}$
22: **return** $M = \{m_1, m_2, ..., m_h\}$

Step 2 - Calculate STL size λ: The analysis in this step is based on the assumption that the frequencies of terms in the whole corpus (i.e. the collection of all terms in the dataset) follow the Zipf distribution. This is true [18] for the collection of documents collected from several online sources: *Myspace*[1], *Twitter*[4], *Slashdot*[3].

We first consider calculating λ for the RA algorithm. Let each object have x number of terms on average. Therefore, the total number of terms (including duplicates) in the whole corpus is Nx. Let p be the rank of a term and $freq(p, Nx)$ denote the frequency of the p^{th} term in the ordered frequency list of a dataset containing Nx terms. The Zipf law states that the frequency of a term is inversely proportional to its rank in the frequency list. Thus the Zipf parameter c (which is collection specific) is given by:

$$c = p\frac{freq(p, Nx)}{Nx} \quad (7)$$

Using Equation 7, the frequency $freq(p, Nx)$ of a term at any arbitrary rank p can be computed which is $\frac{cNx}{p}$.

In our case, each level i node of the R-tree contains on average Nx/N_i terms. Therefore, the frequency of the p^{th} term in the STL of a level i node is $\frac{c_i Nx}{N_i p}$. Similarly the frequency of the p^{th} term in the STL of the query region q^d is $\frac{c_q q^d Nx}{p}$. The above assumes that the exact frequency of each accessed term from any STL is known, which is a property of the RA algorithm.

Note that a top-k algorithm works by computing the threshold value at successive index p which is the sum of all p^{th} frequency values in the STLs that are involved in top-k calculation. Given Nx, q and M, the threshold value at an index p is computed as,

$$\theta(p, q, Nx, M) = \sum_{i=1}^{h} m_i \frac{c_i Nx}{N_i p} \quad (8)$$

The top-k threshold algorithm stops at an index p when the threshold value equals or drops below the k^{th} frequency value in the query region q^d. Therefore the expected list size is computed as,

$$\lambda_{RA}(k, q, Nx, M) = \min_p \{\theta(p, q, Nx, M) \leq \frac{c_q q^d Nx}{k}\} \quad (9)$$

When considering the NRA algorithm, the scan may have to go further than RA; the reason is that when we have accessed the first p terms of each STL, we may only know the partial final frequency of some terms. The NRA algorithm terminates when two conditions hold at the same time: (a) the minimum score of the k-th best term so far, y, is higher than the threshold, and (b) that score is also higher than the maximum score of other partially seen terms. Computing the optimal λ requires knowledge of the correlation of the term frequencies across lists. Instead, we compute a conservative estimation (overestimate), by assuming that we have only seen each term in only one list. Then, the larger MBRs dominate the scores, and hence we can assume that y equals the k^{th} term of the largest MBR, where the largest MBR can be assumed to be one level below the query region q^d; that is, the largest MBR has area q^d/f. Given these assumptions, the conservative estimation of λ for NRA is given by (we further justify the choice of λ in Appendix A):

$$\lambda_{NRA}(k, q, Nx, M) = \min_p \{\theta(p, q, Nx, M) \leq \frac{c_q q^d Nx}{kf}\} \quad (10)$$

6. MULTI-REGION QUERIES

With multi-region kFST queries a user can combine or exclude terms from multiple regions. Consider for example the month before the US elections. It would be interesting to know about the popular terms that appear in social media in some states combined (e.g. the battleground states, say Florida and North Carolina) to see how the public opinion is formed in those states. One can also normalize the term frequencies within each respective state so that larger states do not dominate the top-k calculations. We may also be interested in excluding terms that are popular in "blue" states (e.g. New York and California) so as to identify the terms that are of interest to the republican voters in the battleground states.

Straightforward approach: To solve the multi-region kFST we can simply run the algorithm separately for each region (the R-tree will be traversed multiple times) and then add/subtract the frequencies of the top-k terms in different regions to obtain the terms which have the maximum frequencies combined.

Proposed approach: It is more efficient to compute the multiple regions kFST in a single traversal of the R-Tree. The case where the kFST contains only *included regions* is simple: to find terms which are popular in all these areas we add the term frequencies for STLs belonging to these regions. The more interesting case is when the kFST contains *excluded regions*. If a term t is found in one of the excluded regions' STL, its score is penalized by subtracting t's frequency in that STL. Furthermore, the threshold calculation is also affected. For the RA algorithm, finding t in the excluded region adds zero to the threshold (since we subtract, this is the highest value from the excluded STL). Similarly, for the NRA the *tops* value for this particular STL would be zero (which is the maximum possible value for any of the unseen terms in that list).

Consider the example in Figure 2 assuming that R_3 is an included region while R_4 is an excluded region. At the first iteration, RA accesses the first position in each STL, i.e., terms t_1 and t_2. RA finds these terms in all STLs and computes their scores: $f_{R_Q}(t_1) = 3 - 2 = 1$ and $f_{R_Q}(t_2) = 3 - 1 = 2$. The θ value at this point is $3+0 = 3$. The algorithm proceeds and scans the terms at the second position, t_4 and t_2. The score of t_4 is $3 - 1 = 2$. At this point, the top-2 terms are t_4 and t_2 (ties are broken arbitrarily). The θ value at position 2 is $3 + 0 = 3$ which is higher than the frequencies of t_4 and t_2. At the next position we see no new terms but θ becomes $2 + 0 = 2$ which is no higher than the frequencies of t_4 and t_2. RA ends and t_4 and t_2 are the top-2 terms.

The detailed modifications needed so that RA-STL (Algorithm 3) and NRA-STL (Algorithm 4) work for multiple regions appear in Appendix B. We denote the multi-region algorithm variants by prepending the "MR-" prefix, e.g., MR-RA-STL-Li.

7. EXPERIMENTAL EVALUATION

We proceed with the experimental evaluation results. Table 4 depicts the name convention used for the algorithms presented in the experiments (where x refers to the kind of STLs used).

7.1 Setup

All experiments are performed on a 3.4GHz Intel Core i7-3770 CPU, 16GB RAM machine running Windows 10 OS.

Datasets: For our experiments, we crawled 15,124,195 geotagged, english-based tweets using the Twitter streaming API [4]. The temporal domain covered two years (2012, 2013) while the spatial domain was the whole world. We removed the stop keywords from the tweet text to get meaningful top-k terms. After removing the stop keywords, each tweet has 9 terms on average. We term this as the 15M dataset. To measure scalability performance we also used two artificial datasets that had 50M and 100M tweets. These datasets have the same temporal domain as the 15M dataset; to create the 50M and 100M datasets, for each real tweet we added 4 and 8 (respectively) artificial tweets by changing the original tweet's geolocation .

Index Structure: All tweets in each dataset are indexed by an R-tree. The page (node) size is set to $8KB$ which corresponds to a maximum of 100 entries; the average fanout factor f is 70. We store the STLs of the R-tree nodes in a column family using *Cassandra* 2.0.5. To improve the efficiency of the threshold algorithms (both RA and NRA), we divide each STL into pages of size 250 entries and store each STL page as a separate row in the column family. The row key is the concatenation of the STL identifier and the page index. Furthermore, to facilitate random access on each STL (RA algorithm), we store the terms of a given STL in a separate column family. Here, each term is stored as a separate row, where the row key is the concatenation of the STL identifier and the term. The value is the frequency of that term in this STL.

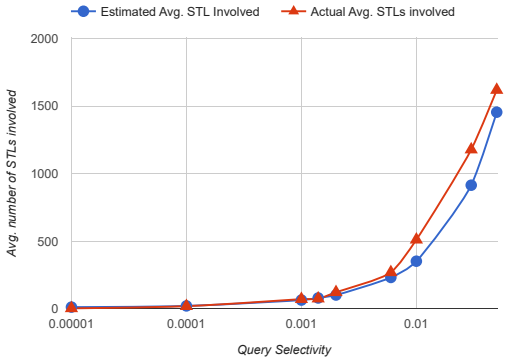


Figure 8: Estimated and Actual average number of STLs involved in the top-k calculation for different query region selectivities for the 15M dataset.

Algorithm *	Description
NRA-STL-x	Sequential access in STL-x
RA-STL-x	Random access in STL-x
MR-NRA-STL-x	Multi-Region NRA-STL-x
MR-RA-STL-x	Multi-Region RA-STL-x
RTreeScan	Baseline 1
POSTGIS	Baseline 2

* x = L or LI or Li or li

Table 4: List of Algorithms.

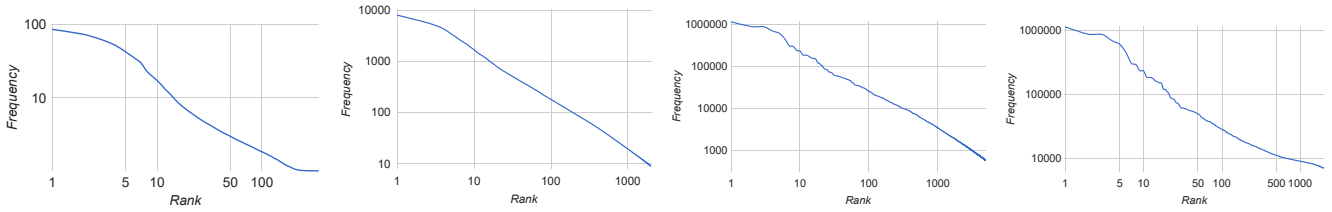
In the first baseline method (*RTreeScan*), we use R-tree to find the tweets that are inside the query region; their term aggregation is performed very fast using a hash map (in main memory).

In the second baseline method (*POSTGIS*), we use the PostGIS [2] spatial database extender to index the tweets. We first run a query to find the tweets that are inside the query region and then we aggregate their term frequency using a hash map residing in main memory.

Query Distribution: For simplicity, the query regions used in the experiments have square faces. The term "query selectivity" denotes the fraction of the total dataset area (or volume) covered by the query. Our datasets cover the total area of the earth, i.e., 196.9 million sq. miles. Hence, our smallest selectivity, which is 0.00001 corresponds to 1970 sq. miles, and so on. For the smaller dataset of 15M tweets, this corresponds to an average of 150 tweets per query, while for our largest selectivity of 0.05 there are about 750,000 tweets on average. For the larger datasets 50M and 100M, our smallest selectivity 0.00001 has 500 and 1000 and our largest selectivity 0.05 has 2500,000 and 5000000 respectively. For each query selectivity the results are averaged over 100 different queries with that selectivity. The default value of k is set to 10 in all experiments except Figure 15 where we consider queries with various values of k.

7.2 Model Validation

We first compare the theoretically estimated number of STLs involved in a query ($\sum_{i=1}^{h} m_i$, where m_i is calculated by Algorithm 7) to the actual number of STLs (averaged over multiple queries).



(a) Level 1 [c = 0.071788] (b) Level 2 [c = 0.10699] (c) Level 3 [c = 0.100089] (d) Level 4 [c = 0.08027]

Figure 9: Calculating the Zipf parameter for different levels of the R-tree (15M dataset).

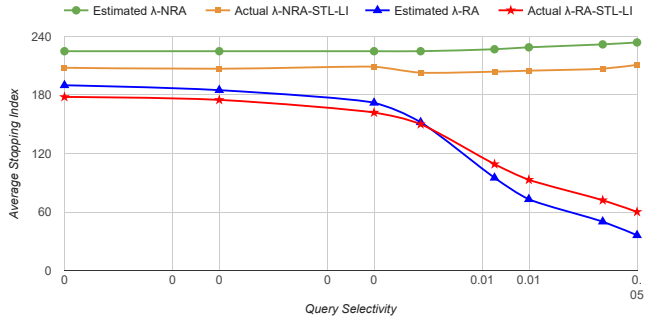


Figure 10: Estimated λ and average stopping points of RA-STL-Li and NRA-STL-Li for different query region selectivities (15M dataset)

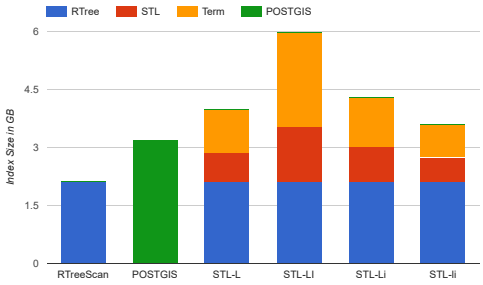


Figure 11: Space requirements (15M dataset).

Figure 8 shows the estimated value and the average number of STL/nodes calculated by Algorithm 2 for different query selectivities using the 15M dataset (this algorithm is used by both RA and NRA). As expected, the number of STLs increases with the query selectivity. Overall the model behaves well (slightly underestimating the actual value); this is to be expected because our analysis assumes uniform distribution for the object locations, while in practice, the tweet locations follow a skewed distribution.

As mentioned in Section 5, our model assumes that the terms in the tweets follow the Zipf distribution and that this also holds for the tweets at each node of the R-tree. Figure 9 depicts the average term frequency vs. rank (p) for the STLs at different levels of the R-tree (level 1 corresponds to the leaf nodes) for the 15M dataset. The slope of each graph corresponds to the Zipf parameter (c) at each level. As it can be seen the values of c are similar across levels (validating our assumption). For computing λ, we set the query region Zipf parameter (c_q) equal to the average c across all levels of the R-tree.

Figure 10 shows the theoretically estimated λ_{RA} and λ_{NRA}, using Equations 9 and 10 respectively, along with the actual average STL list length accessed by the RA-STL-Li and NRA-STL-Li algorithms respectively for the 15M dataset, for various query selectivities.

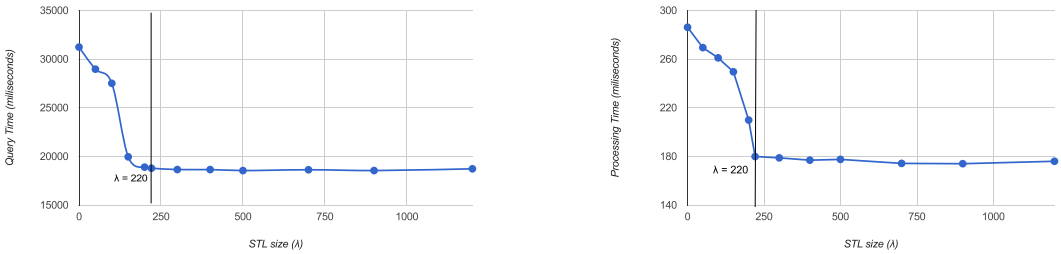
The estimated λ_{RA} follows closely the actual RA-STL prefix length. We further observe that at the lower selectivities the model slightly overestimates, because at these selectivities, the query region is small enough that it typically contains/overlaps only leaf nodes. However, the model does not account for all partially contained nodes and their terms; it thus assumes that the algorithm has access to fewer terms than in reality. In case of RA-STL, when a term is first encountered its exact score is calculated. If fewer terms are seen, the chance that the threshold will be exceeded decreases. In contrast, for higher selectivities the model slightly underestimates, as the query is large enough to contain inner nodes. Our model assumes that an inner node is contained as long as the query region exceeds the inner node size; in that case the model uses a higher level STL. A higher level STL contains terms with higher frequencies dominating the top-k calculation. In reality however, there may be smaller inner nodes not fully contained by the query and thus the RA-STL will access children nodes and involve many lower level STLs. As a result, the top-k calculation finishes later than what the model estimates.

As expected, the estimated λ_{NRA} overestimated the list prefix length, because Equation 10 makes a conservative assumption that a term appears in one STL while in practice, it typically appears in multiple STLs. Nevertheless, the estimated λ_{NRA} closely follows the trend of the actual accessed length.

For the rest of the experiments with the 15M dataset, we pick $\lambda_{RA} = \lambda_{NRA} = 220$, which is given by Equation 10 for λ_{NRA} for middle selectivities. We know that this is an overestimation of the list length for both RA and NRA, so this choice ensures that there is low probability of needing to access more than 220 terms for any list. Figures 12(a),(b) show the query processing time for different values of λ for RA-STL-Li and NRA-STL-Li respectively, using query selectivity 0.002 for the 15M dataset. In both cases the performance initially improves drastically as λ increases and stabilizes when $\lambda > 220$. Similarly, for the 50M and 100M dataset, we chose λ as 430 and 650 respectively.

7.3 STL Approaches Comparison

Index Size: Figure 11 depicts the space requirements for the four approaches (STL-L, STL-LI, STL-Li and STL-li) and the two baselines, RTreeScan and POSTGIS for the 15M dataset. For each method, we show the space needed by the R-tree, the STLs and Term index (the Term index is the Cassandra column store used to facilitate the random accesses (RA); hence it is not needed for the NRA algorithms). Since the R-tree is identical in all STL approaches, its size is the same; this is also the space used by RTreeScan. The POSTGIS approach uses the GiST index for indexing the data. The STL-L approach stores STLs only for the leaf level nodes, thus it requires the least STL storage among all STL-based



(a) RA-STL-Li

(b) NRA-STL-Li

Figure 12: Avg. processing time for different STL sizes (λ) for query selectivity of 0.002 (15M dataset).

Index Structure	15 M	50 M	100 M
STL-Li	3.45	19.64	42.76
RTreeScan	2.13	8.97	19.54
POSTGIS	5.64	22.36	50.79

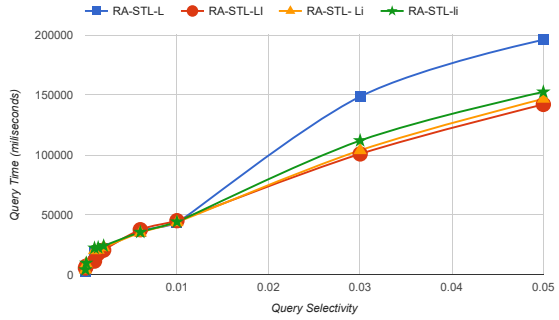
Table 5: Index size for various datasets (in GB).

approaches. At the other end, the STL-LI approach stores full STLs for all nodes and thus uses the largest space. STL-Li replaces the inner lists with partial STLs saving on the STL space; STL-li uses the least space. The Term index space relates to the terms in the STLs used hence it behaves similarly to the STL space. Table 5 shows the space required for STL-Li, RTreeScan and POSTGIS for various datasets for spatio-temporal queries.

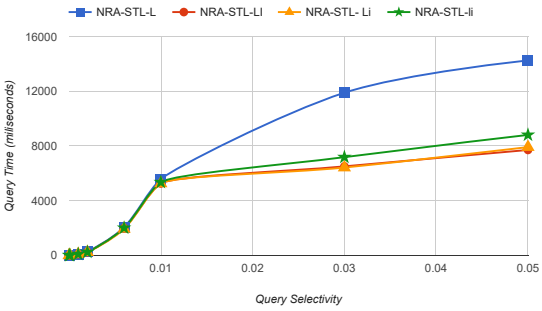
kFST Query Processing: Figures 13a and 13b present the single-region kFST query performance comparison for the four approaches (STL-L, STL-LI, STL-Li and STL-li) using the RA and NRA algorithms respectively for the 15M dataset. In all cases, the query time increases with the query selectivity. Note that, when the query region is less than the MBR size of level 2 nodes, only leaf level STLs are involved in RA-STL-x and NRA-STL-x. For these selectivities all four approaches perform similarly. As the query size increases, some leaf level STLs are replaced by the inner level STL(s) for the STL-Li, STL-li and STL-LI approaches which start to perform better than STL-L. As expected, the full list case (STL-LI) has the best performance; nevertheless, both partial list approaches (STL-Li and STL-li) show similar performance. This is because with $\lambda = 220$ the partial STLs are sufficient to answer the query regions greater than the size of the level 2 nodes (as Figure 12 also showed). Among the partial list approaches, STL-li is slightly slower than STL-Li since it has to compute some leaf lists from scratch.

Given the space and query time trade-offs, the STL-Li is a good compromise for both the RA and NRA algorithms (offering space close to the minimum of STL-L and query times close to the STL-LI). Next, we compare the NRA-STL-Li and RA-STL-Li with RTreeScan and POSTGIS the 15M dataset. The results appear in Figure 14 (note the logarithmic scale on the query time). The NRA-STL-Li consistently outperforms the other methods. Interestingly, the RA-STL-Li is slower than the RTreeScan baseline; the reason is the many random accesses it performs as the number of STLs increases.

We also examined the sensitivity of our approach (NRA-STL-Li) to the value of k. As discussed in Section 4.3, the partial lists assume that k is known in advance. In Figure 15 we assume that λ was computed using $k = 10$ and examine the behavior of the NRA-STL-Li algorithm when the query uses different k (varied from 5 to 20) on the 15M dataset. As it can be seen from the Figure, the partial list algorithm is faster that the baselines for k up to 15.



(a) RA-STL-x



(b) NRA-STL-x

Figure 13: Avg. processing time for different query selectivities (15M dataset).

7.4 Spatio-Temporal Query Experiments

We proceed with a comparison between NRA-STL-Li, RtreeScan for the (more general) spatio-temporal queries. The temporal dimension of the dataset spans over 2 years. To create spatio-temporal queries, we first varied the temporal range from 1 hour (corresponding to selectivity 0.00001) to 1 month (for 0.05) and then computed the needed spatial selectivity so that the total selectivity is the value shown in Figure 16. Please note that We run this experiment on the 100M dataset. Like our previous experiments, NRA-STL-Li outperforms both RTreeScan and POSTGIS.

Scalability: We also examine how our algorithm scales. In this experiment, we use all datasets 15M, 50M and 100M tweets and we compared RTreeScan, POSTGIS and NRA-STL-Li using a spatio-temporal query (constructed as above) with selectivity of 0.002. The result is shown in Figure 17. NRA-STL-Li outperforms both baselines (note the log scale). As we increase the size of the dataset

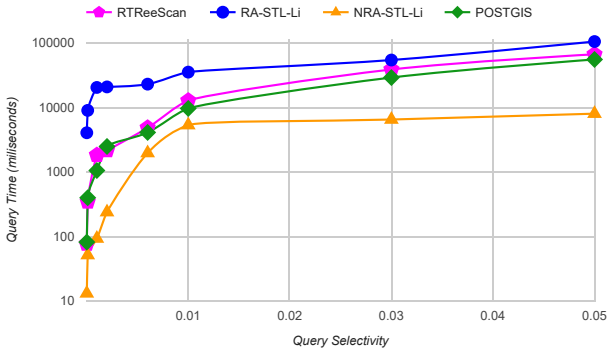


Figure 14: Comparing RA-STL-Li, NRA-STL-Li, POSTGIS and RTreeScan for different selectivity of query regions (15M dataset).

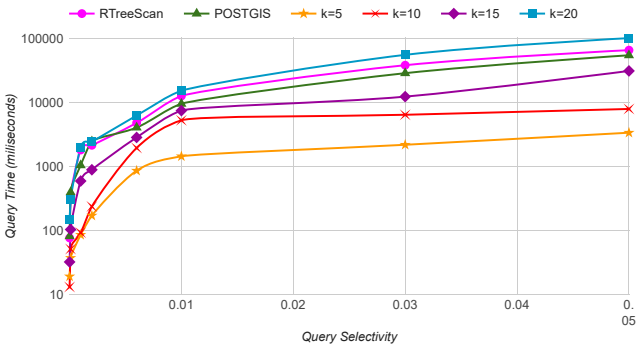


Figure 15: Query performance for different values of k when STLs were created using k=10 (15M dataset).

for the same selectivity, the algorithms have to process more and more data. Hence the query time for all the algorithms increases.

7.5 Discussion

From the experiments presented above we come to the following conclusions: Among the approaches presented, including the baselines RTreeScan and POSTGIS, the best performance (considering query time and space requirements) is given by the NRA-STL-Li algorithm. Here is an ordering of the algorithms from the fastest to the slowest:

NRA-STL-Li > NRA-STL-L > POSTGIS > RTreeScan > RA-STL-Li > RA-STL-L

Note that all the NRA-STL-x algorithms are faster compared to the RA-STL-x algorithms. NRA is faster because our data resides on the disk, which is at least an order of magnitude slower to access randomly, and the depth of access by NRA is only around 2 times more on average. RA could be faster in different scenarios.

Further, the performance advantage of the partial STL-based algorithms (NRA-STL-Li, RA-STL-Li), incurs very small space overhead as compared to the algorithms that do not maintain any internal node STLs (NRA-STL-L, RA-STL-L, RTreeScan) and the POSTGIS approach.

We also found (see Appendix C) that the Multi-Region version of the algorithms MR-RA-STLs-x and MR-NRA-STLs-x perform better compared to the versions where we run RA-STL-x and NRA-STLs-x multiple times, as they traverse the R-tree more efficiently.

As an anecdotal evidence for the usefulness of our system from a user's perspective, we ran a spatio-temporal query which covers the

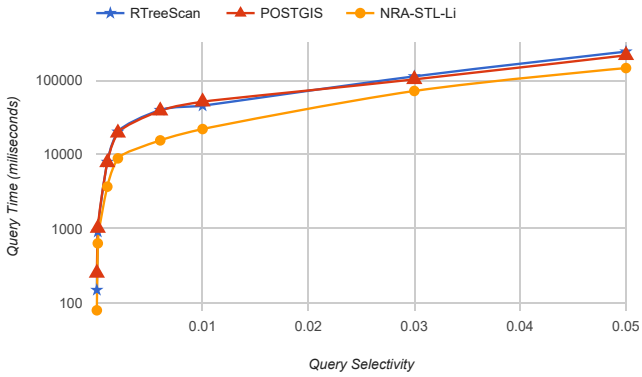


Figure 16: Comparing NRA-STL-Li, RTreeScan and POSTGIS for different spatio-temporal query selectivities (100M dataset).

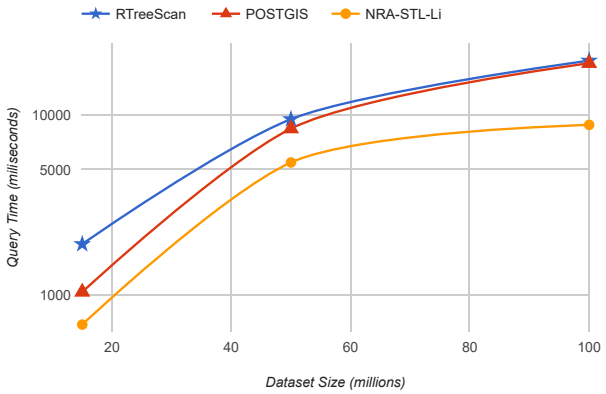


Figure 17: Comparing performance of NRA-STL-Li, RTreeScan and POSTGIS for different dataset sizes and selectivity 0.002.

Catalan region in Spain during the month of October 2013. Among the top-10 frequent tweet terms were terms like 'barcelona', 'madrid', 'xavi', 'fcbarcelona'. These directly correspond to the famous *el clasico* soccer match between Real Madrid and FC Barcelona, which was held on 26 October, 2013.

Acknowledgment

This work was partially supported by NSF grants IIS-1216007, IIS-1447826, IIS-1527984 and IIS-1619463.

8. CONCLUSIONS AND FUTURE WORK

We proposed an indexing scheme that adds sorted term lists (STLs) for fast answering of top-k most frequent term queries over spatio-temporal ranges. Our approach uses a theoretical model to reduce the size of the STLs without sacrificing the query time performance. We presented RA and NRA algorithms that operate on top of the proposed index structures. The NRA algorithm with partial STLs was found to have the best performance (when considering query time and space). We also presented efficient multi-region versions of the algorithms. As future work, we plan to enhance our STL approach with a distributed threshold algorithm (like [8]) so as to process even larger volumes of data. Further, we will study how the proposed indexes can handle high-throughput streaming data.

9. REFERENCES

[1] Myspace, http://myspace.com/.

[2] PostGIS, http://www.postgis.net/.

[3] Slashdot, http://slashdot.org//.

[4] Twitter, http://twitter.com/.

[5] R. Ahuja, N. Armenatzoglou, D. Papadias, and G. J. Fakas. Geo-social keyword search. In *International Symposium on Spatial and Temporal Databases*, pages 431–450. Springer, 2015.

[6] N. Armenatzoglou, R. Ahuja, and D. Papadias. Geo-social ranking: functions and query processing. *VLDB J.*, 24(6):783–799, 2015.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD*, pages 322–331, 1990.

[8] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC*, pages 206–215, 2004.

[9] X. Cao, G. Cong, T. Guo, C. S. Jensen, and B. C. Ooi. Efficient processing of spatial group keyword queries. *ACM Trans. Database Syst.*, 40(2):13, 2015.

[10] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.

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

[12] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[13] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.

[14] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *IEEE ICDE*, pages 656–665, April 2008.

[15] D. Greene. An implementation and performance analysis of spatial data access methods. In *IEEE ICDE*, pages 606–615, 1989.

[16] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, pages 47–57, 1984.

[17] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

[18] G. Inches, M. J. Carman, and F. Crestani. Statistics of online user-generated short documents. In *ECIR*, pages 649–652, 2010.

[19] C. Jonathan, A. Magdy, M. Mokbel, and A. Jonathan. GARNET: A holistic system approach for trending queries in microblogs. *32nd IEEE ICDE*, pages 1251–1262, 6 2016.

[20] T. Lappas, M. R. Vieira, D. Gunopulos, and V. J. Tsotras. On the spatiotemporal burstiness of terms. *Proc. VLDB Endow.*, 5(9):836–847, May 2012.

[21] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *ACM SIGMOD*, pages 401–412, 2001.

[22] A. Magdy, A. Aly, M. Mokbel, S. Elnikety, Y. He, S. Nath, and W. Aref. GeoTrend: Spatial trending queries on real-time microblogs. *ACM GIS Conf.*, 2016.

[23] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung. Efficient aggregation of ranked inputs. In *IEEE ICDE*, pages 72–72, April 2006.

[24] S. Nepal and M. V. Ramakrishna. Query processing issues in image(multimedia) databases. In *IEEE ICDE*, pages 22–29, 1999.

[25] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient processing of top-k spatial preference queries. *PVLDB*, 4(2):93–104, 2010.

[26] A. Skovsgaard, D. Sidlauskas, and C. S. Jensen. Scalable top-k spatio-temporal term querying. In *IEEE ICDE*, pages 148–159, March 2014.

[27] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *PODS*, pages 161–171, 1996.

[28] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *IEEE Trans. Knowl. Data Eng.*, 24(10):1889–1903, 2012.

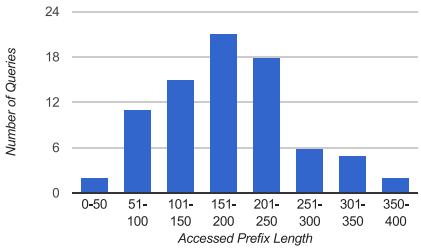
[29] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *IEEE ICDE*, pages 541–552, 2011.

[30] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis. Top-k spatial preference queries. In *IEEE ICDE*, pages 1076–1085, 2007.

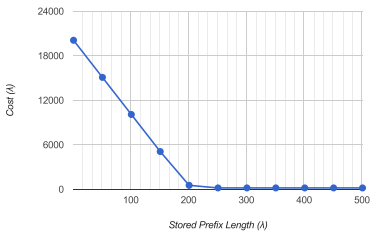
APPENDIX

A. JUSTIFICATION FOR CHOICE OF PRECOMPUTED PREFIX LENGTH

Note that in Equations 9 and 10 we computed the expected prefix length $E(p)$. We will now justify that setting the precomputed prefix length λ to $E(p)$ is an effective choice. Let p be a variable representing the prefix length that a query accesses in an STL. Figure 18a shows the distribution of p for our 15M tweets dataset for query selectivity of 0.002 across all accessed STLs. For our theoretical analysis, we assume that p follows the binomial distribution, which has a similar shape; that is, we have $Prob(p) = \binom{n}{p} \cdot P^p(1-P)^{n-p}$, where n and P are parameters of the binomial distribution, which has mean $E(p) = nP$.



(a) Distribution of accessed prefix lengths



(b) Cost vs. precomputed prefix length

Figure 18: Trade-off between accessed prefix length and execution cost (time).

The expected cost $Cost(\lambda)$ of accessing a partial STL has two

components: (i) the cost of RA on the precomputed prefix length, which is $a \cdot p$, where a is a constant (representing the average cost of accessing an item in the list) and p is the prefix length; (ii) $a \cdot f \cdot (p - \lambda)$ for the access to the f children of the involved nodes when $p > \lambda$ terms are needed. Hence,

$$Cost(\lambda) = a \cdot \sum_{p=1..l} (Prob(p) \cdot p) + a \cdot f \cdot \sum_{p=(\lambda+1)..l} (Prob(p) \cdot (p - \lambda)) \quad (11)$$

Note that λ takes values from 0 to l, where l is the maximum length of the STL if we would store all its terms. To plot the cost function, we select the following concrete values: $a = 1$ (its choice does not affect the shape), $f = 100$ (which is a typical branching factor for R-trees, which we also use in our experiments; increasing it would make the graph more steep), and $l = 450$ (increasing further has no effect as the probability is almost 0 for larger values). For the binomial, we used $P = 0.5$, $n = 400$, so $E(p) = 200$ (different values of P make the binomial curve wider or narrower). Figure 18b shows that there is a clear elbow at $\lambda = 200$, which is also the mean prefix length $E(p)$. Equations 9 and 10 indeed compute $E(p)$ using the properties of the R-tree and the terms' distribution. Figure 12 shows experimentally the same behavior of the cost (time) as a function of λ.

B. ALGORITHMIC MODIFICATIONS FOR MULTI-REGION QUERIES

The modifications needed so that RA-STL (Algorithm 3) works for multiple regions appear in Table 6. The original line 8 needs to run once for each region in $\{S^+, S^-\}$; hence we replace it with 8(a-d). The calculation of θ in line 9 will also have to be modified to accommodate Included and Excluded regions. Line 14 which computes the individual term frequency for leaves should run once for each of the query regions (replaced by 14(a-b)). The calculation of the term frequency f depends on the type of STL the term comes from; thus line 20 is changed accordingly (op_i is an addition operation for STLs from Included Regions and subtraction operation for Excluded Regions).

The following modifications are needed on NRA-STL (Algorithm 4) to support multiple region kFST queries. Line 8 needs to run once for each region in the region list. It is thus replaced with 8(a-d) as shown in Table 7. In Line 9, the calculation of $tops[n]$ will also have to be modified to accommodate the Included and Excluded regions. If the node is in an Included region, its value will be the frequency of the term, otherwise, it will be 0. In lines 16 and 18, we have to modify the calculation of the term frequency f for the each region in the list depending on which list the region belongs to (we add for an Included Region and subtract for and Excluded Region as shown).

Replace line 8 by: 8(a): **for** each R_Q in $\{S^+, S^-\}$ **do** 8(b): $t_e \leftarrow GetTermEntry(n, R_Q, i)$ 8(c): **if** t_e is not null **then** 8(d): **break**
Replace line 9 by: 9(a): **if** n in S^+ **then** $\theta \leftarrow \theta + t_e.Freq$ 9(b): **else if** n in S^- **then** $\theta \leftarrow \theta + 0$
Replace line 14 by: 14(a): **for** each R_Q in $\{S^+, S^-\}$ **do** 14(b): $f' \leftarrow f' + CompTermFreq(n, t_e, R_Q)$
Replace line 20 by: 20(a): $f' \leftarrow f'(op_i) f_{R_Q \cap R'_n}(t)$

Table 6: Changes to Random Access (RA) for Multi-Region kFST.

Replace line 8 by: 8(a): **for** each R_Q in $\{S^+, S^-\}$ **do** 8(b): $t_e \leftarrow GetTermEntry(n, R_Q, i)$ 8(c): **if** t_e is not null **then** 8(d): **break**
Replace line 9 by: 9(a): **if** n in S^+ **then** $tops[n] \leftarrow t_e.Freq$ 9(b): **else if** n in S^- **then** $tops[n] \leftarrow 0$
Replace line 16 by: 16(a): $tke.partialScore \leftarrow tke.partialScore\ (op_i)$ $CompTermFreq(t_e, R_Q)$
Replace line 18 by: 18(a): $tke.partialScore \leftarrow tke.partialScore\ (op_i)\ t_e.Freq$

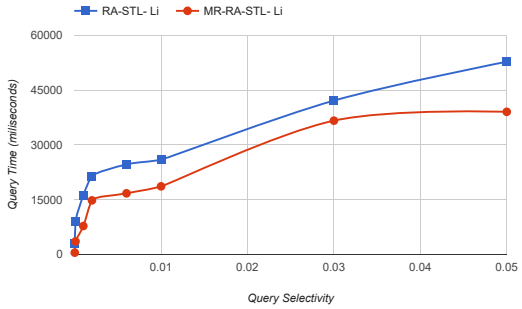
Table 7: Changes to Non-Random Access (NRA) for Multi-Region kFST.

C. MULTI-REGION QUERY EXPERIMENTS

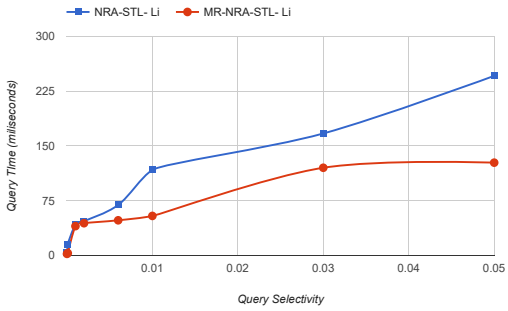
We proceed with the evaluation of the multi-region kFST algorithms. In these experiments we used the 15M dataset. Specifically, we compare the "straight-forward approach" which uses single-region algorithms as modules (denoted as RA-STL-Li and NRA-STL-Li in this experiment), and the optimized multi-region versions (MR-RA-STL-Li and MR-NRA-STL-Li). We compared the algorithms for two regions (both 'included'), where the "straight-forward" approach runs RA-STL-Li or NRA-STL-Li once for each region. Figures 19a and 19b show that the Multi-Region algorithms perform better. This is because a single region algorithm has to run multiple times (in this case twice) and hence traverse the R-tree multiple times. Further, the NRA variants perform better than the RA, as is the case for single-region queries.

Next we consider queries when both 'included' and 'excluded' regions are present. For this experiment, given a selectivity, we randomly select 2 regions as included and another 2 regions as excluded. Figure 20a and Figure 20b depict the comparisons. The MR-RA-STL-Li and MR-NRA-STL-Li approaches are again faster.

In terms of the number of MBR accesses, if a single-region query accesses n MBRs, the multi-region query accesses at most $m * n$ MBRs, where m is the number of regions. The best scenario is when there is very large overlap in which case it accesses close to n MBRs (as the R-tree access paths overlap as well). In terms of the number of STLs involved in the query, if a single region query reads l STLs, the multi-region query reads $m * l$ STLs in the worst case and around l in the best case (when the m regions have very



(a) RA-STL-Li vs MR-RA-STL-Li.



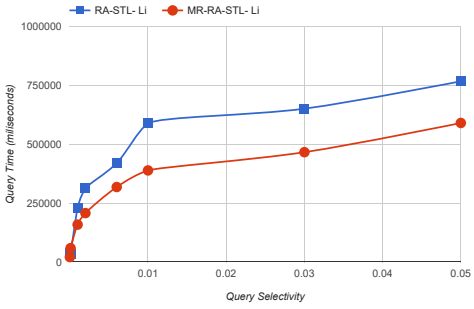
(b) NRA-STL-Li vs MR-NRA-STL-Li.

Figure 19: Multi-region query processing using 'included' regions.

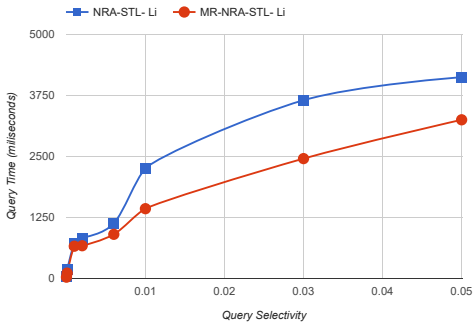
large overlap to each other). The exact overhead depends on how far the regions are from each other.

D. COMPARISON TO APPROXIMATE SOLUTIONS

We finally compare our approach with the approximate solution of AFIA [26]. AFIA keeps $k+1$ items in their materialized lists in each grid cell. We emulate its performance by keeping only $k+1$ items in all of our STLs. The top-k algorithm is forced to terminate if it crosses the $k+1$ STL term. We then compare the returned top-k terms with the (exact) solution that our algorithm would provide. Figure 21 shows the average percentage of error for the approximate approach with $k = 10$ (using the 15M dataset). The error is computed as $missed/k$ where $missed$ corresponds to the number of terms in the correct answer that are not included in the approximate answer (i.e., we do not consider the position of a term in the returned answer). The error increases with the selectivity since the number of STLs involved in the calculation increases, and so does the number of lists for which we have to access beyond the k+1st term.



(a) RA-STL-Li vs MR-RA-STL-Li



(b) NRA-STL-Li vs MR-NRA-STL-Li

Figure 20: Multi-region query processing using 'included' and 'excluded' Regions.

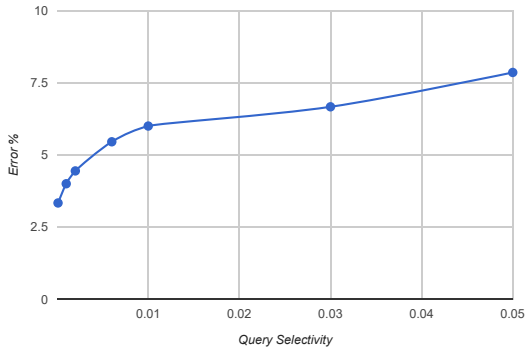


Figure 21: Percentage of error for the approximate solution using different selectivities (15M dataset).