UNIVERSITY OF CALIFORNIA
RIVERSIDE

Top-k and Reverse Top-k Queries Over Spatio-Temporal Ranges

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Pritom Ahmed

December 2020

The Dissertation of Pritom Ahmed is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First of all, I am grateful to my advisor Dr. Vassilis Tsotras; without whose help, I would not have reached this milestone. I want to take this opportunity to thank him for being a constant source of hope and motivation during my PhD.

I want to thank my collaborators Dr. Vagelis Hristidis and Dr. Ahmed Eldawy, for their continuous guidance during my PhD. Both of them have been very helpful and patient with me during this time. I would also like to thank my committee member Dr. Eamonn Keogh for providing useful suggestions and feedback.

The text of this dissertation, in part or in full, is a reprint of the material as it appears in the Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 2017) titled “Efficient Computation of Top-k Frequent Terms Over Spatio-Temporal Ranges” and the arXiv.org preprint titled “Reverse Spatial Top-k Keyword Queries”. The co-author (Dr. Vassilis Tsotras) directed, co-written, and supervised the research, forming the foundation for this dissertation. The co-authors Dr. Vagelis Hristidis and Dr. Ahmed Eldawy reviewed the paper and provided useful suggestions and feedback throughout the research.

This research was partially supported by NSF grants: IIS-1527984, IIS-1838222 and IIS-1901379. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation.

I dedicate this work to my loving wife, Afrin Hossain, for her support and continuous encouragement. I also dedicate this work to my parents Shabbir Ahmed, Rokshan Ara Sheuly, and my aunt Parveen Ahmed who always loved me unconditionally and made me who I am. Without them in my life, I would not be where I am today.

ABSTRACT OF THE DISSERTATION

Top-k and Reverse Top-k Queries Over Spatio-Temporal Ranges

by

Pritom Ahmed

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2020
Dr. Vassilis J. Tsotras, Chairperson

The wide availability of tracking devices has drastically increased 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 thesis, we first study a basic analytics query on geotagged data, which we refer to as the Top-k Frequent Spatiotemporal Terms (kFST) query. Given a spatiotemporal region, kFST finds the most frequent terms among the social posts in that region. While there has been prior work on keyword search and on group keyword search on spatial data, our problem is different in that it returns keywords and aggregated frequencies as output, instead of having the keyword(s) 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. This thesis proposes an index structure and algorithms to answer kFST queries. Our index structure employs an R-tree augmented by top-k sorted term lists, 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. A detailed experimental evaluation of the proposed methods' performance as compared to baselines using real datasets, shows its efficiency and better performance.

Next, we study the reverse problem, namely, the Reverse Spatial Top-k Keyword (RSK) query: given a query term q, an integer k, and a neighborhood size l, find the neighborhoods of that size where q is in the top-k most frequent terms among the social posts in those neighborhoods. An obvious approach would be to partition the dataset with a uniform grid structure of cell size l (a temporal window, a spatial square, or a spatiotemporal cube) and identify the cells where this term is in the top-k most frequent keywords. However, this answer would be incomplete since it only checks for neighborhoods that are perfectly aligned to the grid. What makes the problem challenging challenging is that the complete answer should identify neighborhoods placed anywhere in the search space. Furthermore, for every neighborhood that is an answer, one can easily create another neighborhood with the same data points that are also an answer (by merely moving this neighborhood in any direction as long as it contains the same set of data points). In particular, there are infinitely many such answers. Instead, we identify contiguous regions where any point in the region can be the center of a neighborhood that satisfies the query. We propose an index structure (that employs a uniform grid augmented by materialized sorted term lists) and algorithms to answer the RSK query efficiently. We apply various optimizations that drastically improve query latency against the baseline. We also provide a theoretical model to choose the optimal cell size to minimize query latency. We further examine an approximate version of the RSK, which leads to faster query processing. Extensive experimental performance evaluation of the proposed methods using real Twitter datasets shows the efficiency of our optimizations and the accuracy of the proposed theoretical model.

Finally, we examine the Reverse Spatial Top-k Snapshot Query (RSKSQ), which like the RSK query, identifies areas where a keyword is popular, however, it operates over the Twitter data stream. Given a query term q, an integer k, a neighborhood size l, a time window W, and a refresh rate r, we find the neighborhoods of size l where q is in the top-k most frequent terms among the tweets in those neighborhoods. To answer RSKSQ queries, we run the RSK query over a snapshot of a fixed-sized time window (W) of the most recent tweets, i.e., tweets posted in the last W minutes and refresh the results every r seconds. To implement this approach we use a Ring Buffer B of a fixed size to keep track of the latest tweets. If B becomes full, we discard the oldest tweets. We use an index structure consisting of a uniform grid augmented by materialized lists of term frequencies and a filter-refinement-based RSK query processing algorithm optimized for fast updates to find the answers. We have implemented a system that provides the results of RSKSQ refreshed every r seconds using a desktop application based on ArcGIS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Web users and content are increasingly being geopositioned, and increased focus is being given to serving local content in response to web queries. Several online social media, such as Twitter, Instagram, Foursquare and Facebook, allow users to geotag their social posts. For example, in Twitter[5], users express their thoughts in short messages (*tweets*), and each tweet may be associated with a geolocation, which denotes the originating location of the tweet. This creates novel data analytics problems, such as detecting popular topic trends [42], most frequent trajectories [66], etc. In this thesis, we investigate top-k and reverse top-k on geotagged and time-stamped 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 4.2 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 [35]. 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 [28] 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 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.

Next, we focus on the *reverse* problem: given a keyword, we want to find the spatial (or temporal) regions where this keyword is in the top-k most frequent keywords. This query has many applications, and depending on the application different query sizes or time windows are preferable. Consider an advertiser who wants to monitor Twitter posts and identify neighborhoods where a particular product is among the top-k terms discussed. Smaller result areas (say few blocks in size) may be preferable, where electronic billboards can be utilized, to advertise a new product or offer coupons based on the expressed interest in those areas. Location based social media ads can also be instantly purchased. On the other hand, a political candidate's campaign may be interested in identifying larger areas (so that a political rally can be organized) where a specific topic is popular/unpopular. In this application, posts from a wider time window may be considered (the time window is not an explicit parameter in our problem, as it determines the posts collection size; we consider different collection sizes in our experiments). As shown by these examples, in addition to the query term and its importance, the neighborhood size should also be a query parameter.

The problem is challenging because of the large number of possible neighborhoods (which is $O(N^2)$ for N posts). Instead of searching the whole space, we propose an (exact) algorithm that uses a *filtering* step to prune the search space (without missing any answers) and a scan-based *refinement* step to find the answers in the resulting pruned space. We use a grid-based index structure augmented with a materialized sorted term list at each cell to avoid repeated processing of the tweets during query time. To further minimize the RSK query latency, we propose a theoretical model that estimates the optimal grid index cell size. Nevertheless, the refinement step can be slow because of the sheer number of neighborhoods it has to process to find all the answers. Thus we also explore a restricted version of the problem (RSKR) that limits the possible answers to the cells of a query provided grid. In addition to an exact solution, for the RSKR query, we present faster but approximate algorithms where we restrict the number of neighborhood checks using a budget. The proposed algorithms for RSK and RSKR are highly parallelizable. To take advantage of parallelism, we propose a slicing technique that enables distributing the workload of the refinement step among different nodes and thus further reduce the query latency.

Finally, We present the *Reverse Spatial Top-k Snapshot Query (RSKSQ)*, which operates over the Twitter data stream. We implement it by running the Reverse Spatial Keyword (RSK) query on geo-tagged posts that arrived in time window of W, and refresh the result every r time units. We present a working system using live Twitter stream providing users with options to choose the size of the sliding window as well as the refresh rate i.e., the rate at which the result will be updated. The system is build as a desktop application based on ArcGIS [61].

The rest of the thesis is organized as follows: in chapter 2, we present our work on finding top-k most frequent terms over spatio-temporal ranges (kFST). Chapter 3 describes our approach and findings while answering the reverse problem, i.e., asnwering reverse spatial top-k keyword queries (RSK). Next, in chapter 4, we discuss a working system we developed answering RSKSQ, a snapshot reverse spatial top-k query. Finally, in chapter 5, we summarize our findings and discuss some possible future directions for the research presented in this thesis.

Chapter 2

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

2.1 Introduction

In this chapter, 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 4.2 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 [35]. 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 [28] 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 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 this chapter 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 chapter is organized as follows: Section 4.2 discusses related work, while Section 4.3 formulates the problem. Section 2.4 describes our index structure and Section 2.5 presents the model used to estimate the STL size. We discuss an extension of our algorithms to accommodate multiple query regions in Section 2.6. Our indexing scheme and algorithms are evaluated in Section 2.7 while conclusions appear in Section 2.8.

2.2 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) [47]. 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 [30, 21]. 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 [18, 24]. The work in [78] examines jointly processing multiple top-k spatial keyword queries, while the top-k spatial keyword query for continuously moving objects is studied in [79]. [45] 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, [17] 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 [12]. Again, the problem is different than kFST as posts or users are returned and not terms. Several geo-social query variants are examined in [11]. 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 [11] 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, Figure3.6). 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 [82] 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 [63]. 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 [52, 65] 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 [52] 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 [65] 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 [42] addresses various trending queries on microblogs; 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.

2.3 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 2.7 we also provide performance results under spatio-temporal ranges).

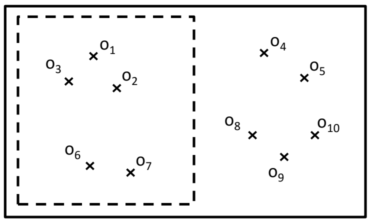
Let $V = \{\cup_{o \in \mathcal{D}} o.Terms\}$ be the vocabulary with all terms. Consider a dataset with 10 objects whose locations in 2D space are shown in Figure 3.1(a); the terms of these objects are shown in Figure 3.1(b). The vocabulary $\{\cup_{i=1}^{9} t_i\}$ contains 9 terms.

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 3.1. The dotted region in Figure 3.1(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



(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}	$\{t_2, t_6, t_7\}$

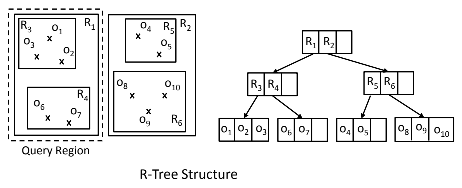
(b) Terms

Figure 2.1: Sample dataset containing 10 objects, (a) shows the locations and (b) shows the terms of the objects.

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.

2.4 Proposed Index Structure and Algorithms

We assume that the full set $\mathcal{D}$ of posts is large and stored on disk. Figure 2.2(a) shows the *baseline* approach that indexes the posts with a multidimensional R-tree [35]. 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 2.1.



(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 2.2: Spatial R-Tree for the sample dataset in Figure 3.1 and leaf level STLs

2.4.1 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 2.2(b) shows the STLs for the two leaf nodes R_3 and R_4 of the R-tree in Figure 2.2(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) [28, 56]. Note that RA is a modified (improved) version of TA. The specific improvements are presented in Section 2.4.4.

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 3.1(a). Based on the R-tree in Figure 2.2(a), the two leaf nodes R_3 and R_4 are fully contained in the query region. Therefore RA executes on these two nodes' STLs (Figure 2.2(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

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 2.1: Different index types.

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: $\mathcal{E} \leftarrow RA\text{-}STL(\mathcal{N}, R_Q, k)$ / $NRA\text{-}STL(\mathcal{N}, R_Q, k)$
3: $\mathcal{T} \leftarrow \emptyset$
4: for each term entry $t_e$ in $\mathcal{E}$ do
5:     $\mathcal{T} \leftarrow \mathcal{T} \cup t_e.Term$
6: return $\mathcal{T}$
```

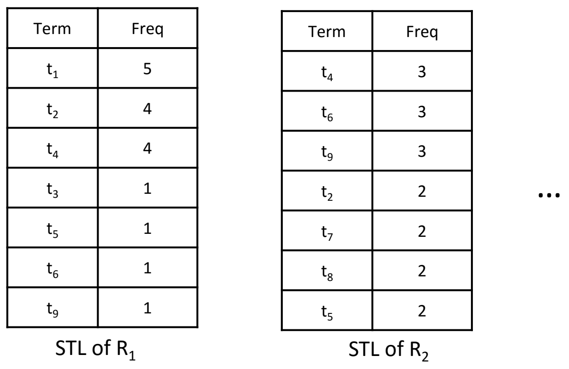
for that term (for details see [28]) and stops when it finds k terms whose Worst Scores are higher or equal to the threshold θ value.

2.4.2 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 2.14a, 2.14b in the experimental evaluation). One solution is to enhance our index structure adding STLs to all inner level nodes of the R-tree. Figure 2.3 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*.

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:      if $R_Q \cap R_n \neq \emptyset$ then
3:          return $\{n\}$
4:      else
5:          return $\emptyset$
6:  if $R_n \subseteq R_Q$ then
7:      return $\{n\}$
8:  else
9:      $\mathcal{N} \leftarrow \emptyset$
10:     for each child $c$ of $n$ do
11:         $\mathcal{N} \leftarrow \mathcal{N} \cup FindCandidateNodes(R_Q, c)$
12:     return $\mathcal{N}$
```



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 2.3: 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

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
 1: $\mathcal{E} \leftarrow \emptyset$, $i \leftarrow 0$
 2: repeat
 3:     $\theta \leftarrow 0,\ f \leftarrow 0$
 4:     for each node $n$ in $\mathcal{N}$ do
 5:         $L \leftarrow getSTL(n)$
 6:         $t_e \leftarrow i^{th}$ term entry in $L$
 7:         if $t_e$ is null and $NotIsLeaf(n)$ then
 8:             $t_e \leftarrow GetTermEntry(n, R_Q, i)$
 9:         $\theta \leftarrow \theta + t_e.Freq$
10:         $t \leftarrow t_e.Term$
11:         if $t$ has not been seen yet then
12:             $f' \leftarrow 0$
13:             if $isLeaf(n)$ then
14:                 $f' \leftarrow ComputeTermFreq(t_e, R_Q)$
15:             else
16:                 $f' \leftarrow t_e.Freq$
17:             for each node $n'$ in $\mathcal{N}$ do
18:                 if $n' \neq n$ then
19:                     do random access for term $t$ on STL of $n'$ and compute $f_{R_Q \cap R'_n}(t)$
20:                     $f' \leftarrow f' + f_{R_Q \cap R'_n}(t)$
21:             $\mathcal{E} \leftarrow \mathcal{E} \cup \{< t, \emptyset, f' >\}$
22:     $f \leftarrow$frequency of the $k^{th}$ term entry in $\mathcal{E}$
23:     $i \leftarrow i + 1$
24: until $\theta > f$
25: return top-$k$ term entries in $\mathcal{E}$ with highest frequencies
```

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 3.1(a). Based on the R-tree in Figure 2.2(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.

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
 1: $SortedTopKElements \leftarrow \emptyset$, $i \leftarrow 0$
 2: repeat
 3:   $\theta \leftarrow 0, f \leftarrow 0$
 4:   for each node $n$ in $\mathcal{N}$ do
 5:     $L \leftarrow getSTL(n)$
 6:     $t_e \leftarrow i^{th}$ term entry in $L$
 7:     if $t_e$ is null and $NotIsLeaf(n)$ then
 8:       $t_e \leftarrow GetTermEntry(n, R_Q, i)$
 9:     $tops[n] \leftarrow t_e.Freq$
10:     $t \leftarrow t_e.Term$
11:     if $t$ has not been seen yet then
12:       $SortedTopKElements \leftarrow tke(t)$
13:     else
14:       $tke \leftarrow SortedTopKElements(t)$
15:     if $isLeaf(n)$ then
16:       $tke.partialScore \leftarrow tke.partialScore + ComputeTermFreq(t_e, R_Q)$
17:     else
18:       $tke.partialScore \leftarrow tke.partialScore + t_e.Freq$
19:     $SortedTopKElements \leftarrow tke(t)$
20:   $\theta \leftarrow \sum_n tops[n]$
21:   $i \leftarrow i + 1$
22: until $SortedTopKElements[k].worstScore \geq \theta$
23: return top-$k$ term entries in $\mathcal{E}$ with highest frequencies
```

2.4.3 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 2.4 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 2.5). Using this λ value we shrink the size of each inner level STL, which reduces the overall space.

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 $R_n$
1: $\mathcal{N} \leftarrow \emptyset$
2: for each child $c$ of $n$ do
3:     $\mathcal{N} \leftarrow \mathcal{N} \cup c$
4: $\mathcal{E} \leftarrow (N)RA - STL(\mathcal{N}, R_Q, i)$
5: return the $i^{th}$ term entry in $\mathcal{E}$
```

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 $f$
```

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*.

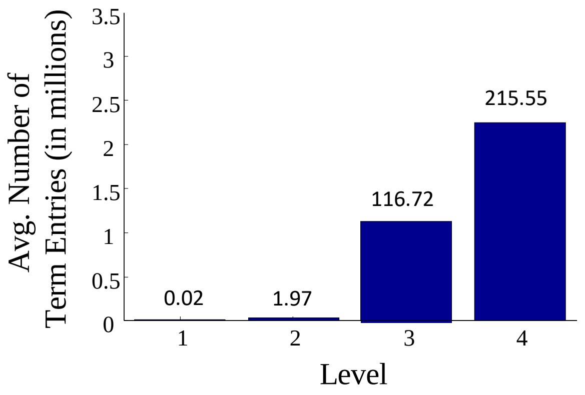


Figure 2.4: Level vs. avg. number of term entries per full STL. The number on top of each bar is the avg. size of that level STL in MB.

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 respectively (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

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 2.2: Changes on RA-STL to reduce random accesses.

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 2.5). 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 2.16) we show that the proposed algorithm performs well for up to 50% larger k than the one originally provided.

2.4.4 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 2.2 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 2.7.1. The results appear in Figure 2.5 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 this chapter, all RA-STL algorithms use the optimized approach.

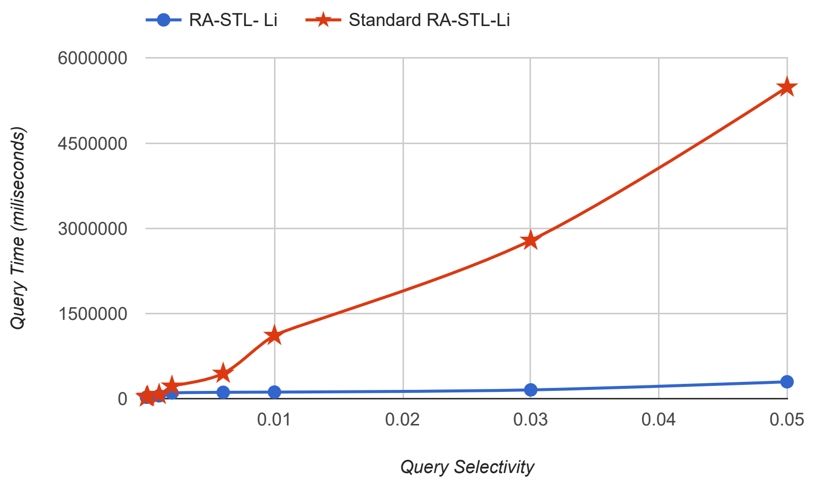


Figure 2.5: Comparison between standard RA-STL-Li and our optimized RA-STL-Li.

Optimizing NRA-STL: After implementing the NRA-STL algorithm, we observed that the CPU time required for sorting the $SortedTopKElements$ is high. We thus applied the following optimizations. (i) **Reduce Buffer Size :** The work in [53] 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 [37] 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 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 2.6 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.

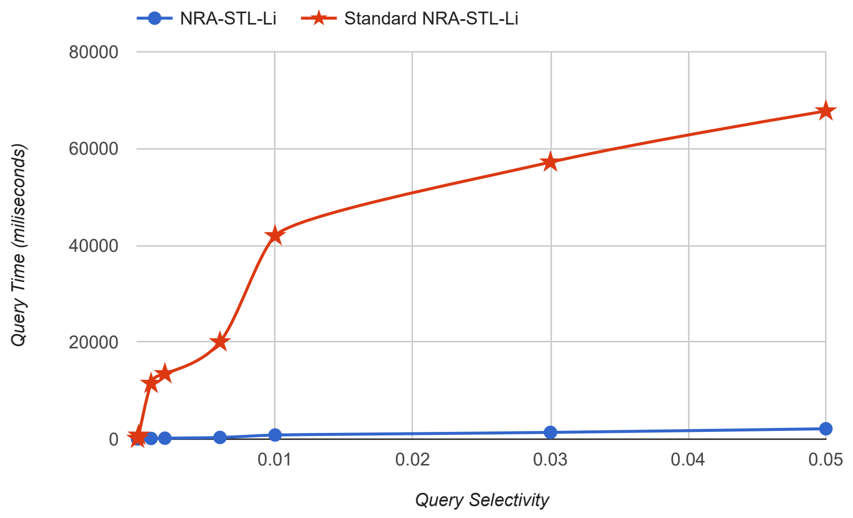


Figure 2.6: Comparison between standard NRA-STL-Li and our optimized NRA-STL-Li.

2.5 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 2.4.3 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

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 2.3: Model Parameters

top-k calculation from level i; h is the height of the R-tree. Using M, we calculate λ in the second step.

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 2.3. 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 [67].

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,

$$N_i = \frac{N}{f^i} \tag{2.1}$$

The height h of the R-tree is calculated as [67, 34],

$$h = 1 + \lceil \log_f \frac{N}{f} \rceil \tag{2.2}$$

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

$$s_i = (f^{1/d} - 1).\frac{1}{(N_{i-1})^{1/d}} + s_{i-1} \tag{2.3}$$

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 [67].

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 [67],

$$intersect(N_i, s_i, q) = N_i.(s_i + q)^d \tag{2.4}$$

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 [67]). The next analysis describes how we can estimate the number of contained nodes given the values N_i, s_i and q.

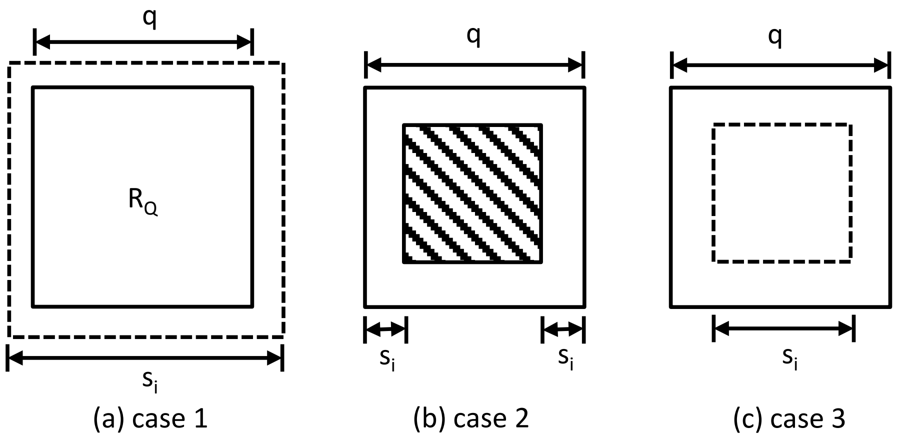


Figure 2.7: Case analysis of s_i and q

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 2.7(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 2.7(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,

$$a_i = (intersect(N_i, s_i, q-2s_i)^{1/d} - 1).\frac{1}{(N_i)^{1/d}} + s_i \tag{2.5}$$

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 2.7(c)). This assumption is a reasonable property for a good R-tree [15]. 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,

$$m_1 = intersect(N_1, s_1, q) - \sum_{i=2}^{h} m_i f^{i-1} \tag{2.6}$$

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.

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

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
2: $s_0 \leftarrow 0$
3: for $i \leftarrow 1$ to $h$ do
4:     Calculate $N_i$ and $s_i$ using Equations 2.1 and 2.3 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 2.6
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 2.5
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 $M = \{m_1, m_2, ..., m_h\}$
```

the Zipf distribution. This is true [39] for the collection of documents collected from several online sources: *Myspace*[2], *Twitter*[5], *Slashdot*[4].

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:

$$c = p\frac{freq(p, Nx)}{Nx} \tag{2.7}$$

Using Equation 2.7, 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,

$$\theta(p, q, Nx, M) = \sum_{i=1}^{h} m_i \frac{c_i Nx}{N_i p} \tag{2.8}$$

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,

$$\lambda_{RA}(k, q, Nx, M) = \min_{p}\{\theta(p, q, Nx, M) \leq \frac{c_q q^d Nx}{k}\} \tag{2.9}$$

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:

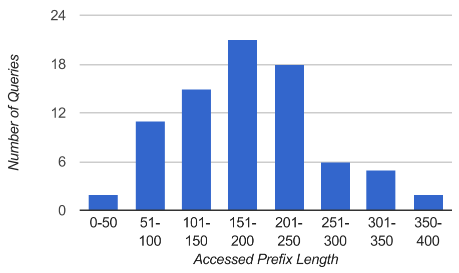
$$\lambda_{NRA}(k, q, Nx, M) = \min_p \{\theta(p, q, Nx, M) \leq \frac{c_q q^d Nx}{kf}\} \tag{2.10}$$

We further justify the choice of λ in in Section 2.5.1.

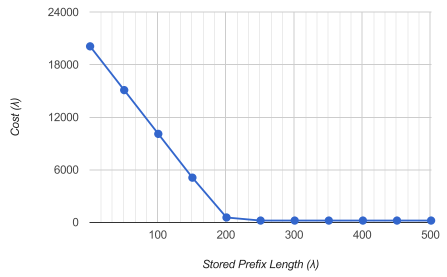
2.5.1 Justification for Choice of Precomputed Prefix Length

Note that in Equations 2.9 and 2.10 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 2.8a 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 $E(p) = nP$.

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



(a) Distribution of accessed prefix lengths



(b) Cost vs. precomputed prefix length

Figure 2.8: Trade-off between accessed prefix length and execution cost (time).

the f children of the involved nodes when $p > \lambda$ terms are needed. Hence,

$$\begin{aligned} Cost(\lambda) = a \cdot \sum_{p=1..l} (Prob(p) \cdot p) + \\ a \cdot f \cdot \sum_{p=(\lambda+1)..l} (Prob(p) \cdot (p - \lambda)) \end{aligned} \tag{2.11}$$

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 2.8b shows that there is a clear elbow at $\lambda = 200$, which is also the mean prefix length $E(p)$. Equations 2.9 and 2.10 indeed compute $E(p)$ using the properties of the R-tree and the terms' distribution. Figure 2.13 shows experimentally the same behavior of the cost (time) as a function of λ.

2.6 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 2.2 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 Section 2.6.1. We denote the multi-region algorithm variants by prepending the "MR-" prefix, e.g., MR-RA-STL-Li.

2.6.1 Algorithmic Modifications for Multi-Region Queries

The modifications needed so that RA-STL (Algorithm 3) works for multiple regions appear in Table 2.4. 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 2.5. 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 2.4: 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 2.5: Changes to Non-Random Access (NRA) for Multi-Region kFST.

2.7 Experimental Evaluation

We proceed with the experimental evaluation results. Table 2.6 depicts the name convention used for the algorithms presented in the experiments (where x refers to the kind of STLs used).

2.7.1 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 geo-tagged, english-based tweets using the Twitter streaming API [5]. 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.

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 [3] 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

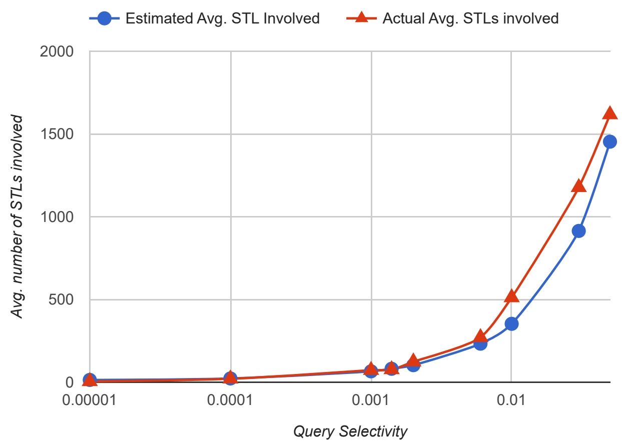
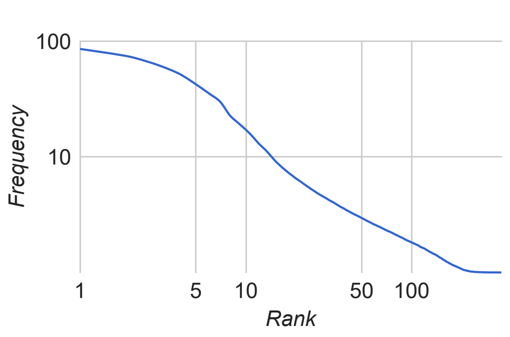


Figure 2.9: Estimated and Actual average number of STLs involved in the top-k calculation for different query region selectivities for the 15M dataset.

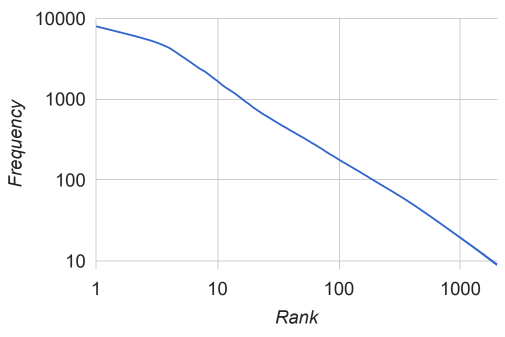
Algorithm *	**Description**
NRA-STL-x	Sequential access in STL-x
RA-STL-x	Random access in STL-x
MR-NRA-STL-x	Multi-Region NRA-STL-x
MR-RA-STL-x	Multi-Region RA-STL-x
RTreeScan	Baseline 1
POSTGIS	Baseline 2

Table 2.6: List of Algorithms (x = L or LI or Li or li).

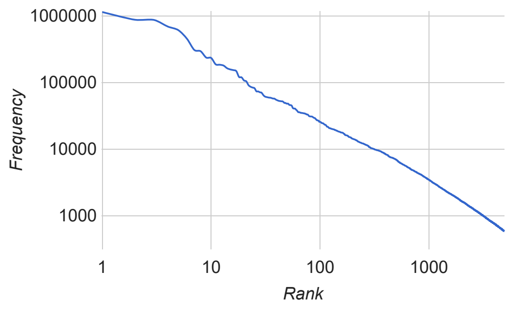
for our largest selectivity of 0.05 there are about 750,000 tweets on average. For the larger datasets 50M and 100M, our smallest selectivity 0.00001 has 500 and 1000 and our largest selectivity 0.05 has 2500,000 and 5000000 respectively. For each query selectivity the results are averaged over 100 different queries with that selectivity. The default value of k is set to 10 in all experiments except Figure 2.16 where we consider queries with various values of k.



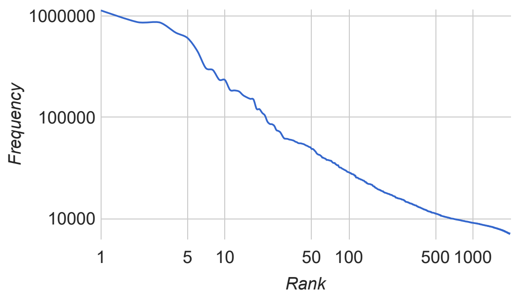
(a) Level 1 [c = 0.071788]



(b) Level 2 [c = 0.10699]



(c) Level 3 [c = 0.100089]



(d) Level 4 [c = 0.08027]

Figure 2.10: Calculating the Zipf parameter for different levels of the R-tree (15M dataset).

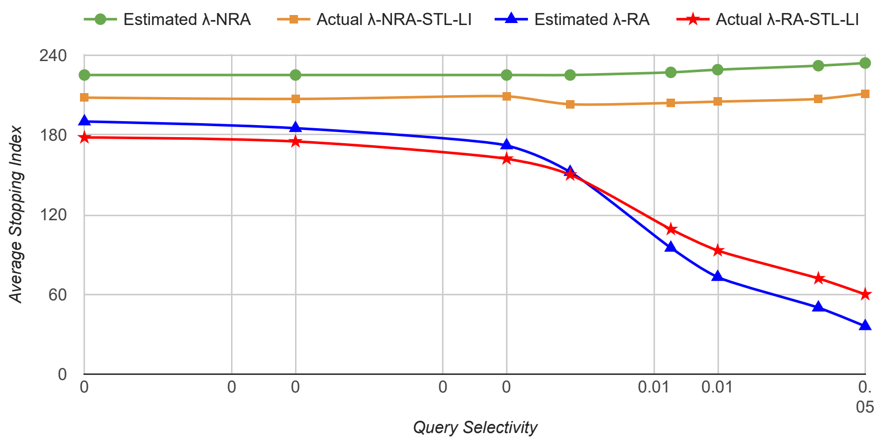


Figure 2.11: Estimated λ and average stopping points of RA-STL-Li and NRA-STL-Li for different query region selectivities (15M dataset).

2.7.2 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). Figure 2.9 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 2.5, 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 2.10 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 2.11 shows the theoretically estimated λ_{RA} and λ_{NRA}, using Equations 2.9 and 2.10 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

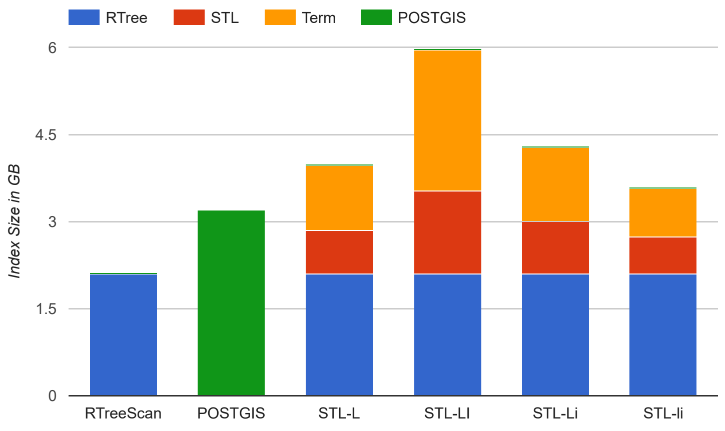
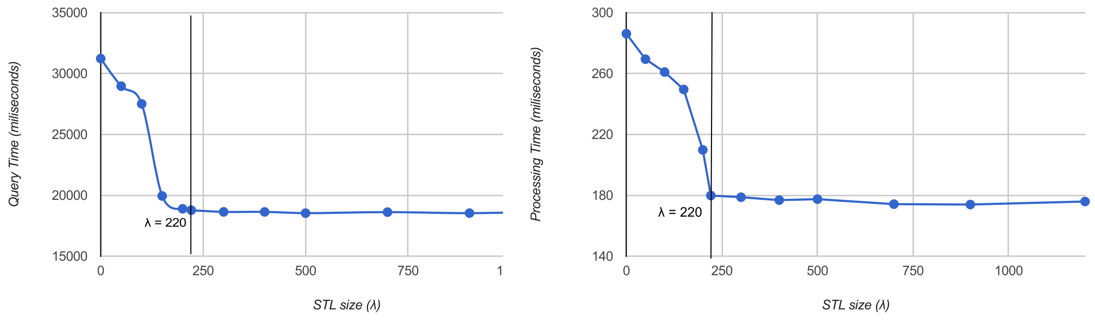


Figure 2.12: Space requirements (15M dataset).



(a) RA-STL-Li

(b) NRA-STL-Li

Figure 2.13: Avg. processing time for different STL sizes (λ) for query selectivity of 0.002 (15M dataset).

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 2.10 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 2.10 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 2.13(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

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 2.7: Index size for various datasets (in GB).

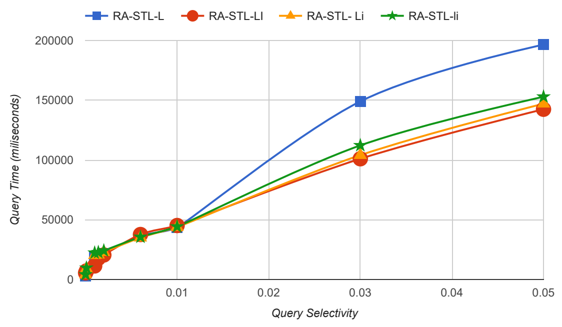
stabilizes when $\lambda > 220$. Similarly, for the 50M and 100M dataset, we chose λ as 430 and 650 respectively.

2.7.3 STL Approaches Comparison

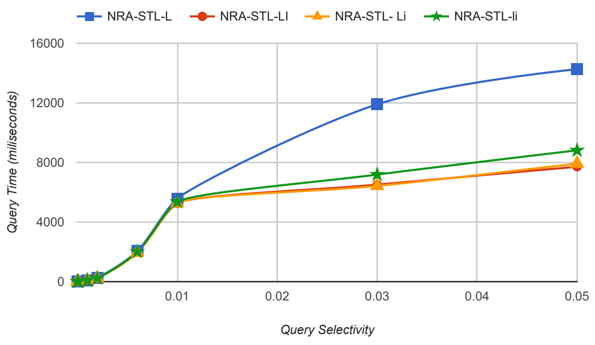
Index Size: Figure 2.12 depicts the space requirements for the four approaches (STL-L, STL-LI, STL-Li and STL-li) and the two baselines, RTreeScan and POSTGIS for the 15M dataset. For each method, we show the space needed by the R-tree, the STLs and Term index (the Term index is the Cassandra column 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 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 2.7 shows the space required for STL-Li, RTreeScan and POSTGIS for various datasets for spatio-temporal queries.

kFST Query Processing: Figures 2.14a and 2.14b 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 2.13 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.



(a) RA-STL-x



(b) NRA-STL-x

Figure 2.14: Avg. processing time for different query selectivities (15M dataset).

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 2.15 (note the logarithmic scale on the query time).

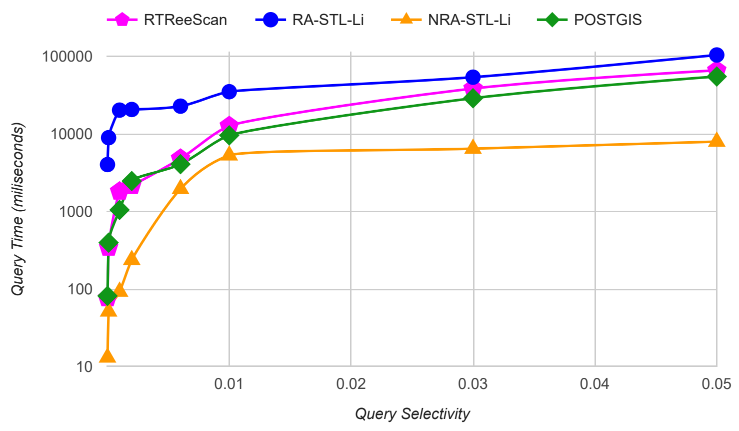


Figure 2.15: Comparing RA-STL-Li, NRA-STL-Li, POSTGIS and RTreeScan for different selectivity of query regions (15M dataset).

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 2.4.3, the partial lists assume that k is known in advance. In Figure 2.16 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.

2.7.4 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

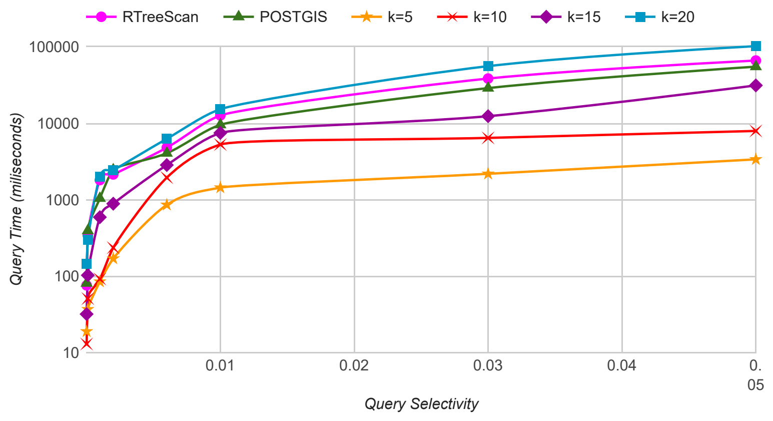


Figure 2.16: Query performance for different values of k when STLs were created using k=10 (15M dataset).

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 2.17. 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 2.18. NRA-STL-Li outperforms both baselines (note the log scale). As we increase the size of the dataset for the same selectivity, the algorithms have to process more and more data. Hence the query time for all the algorithms increases.

2.7.5 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

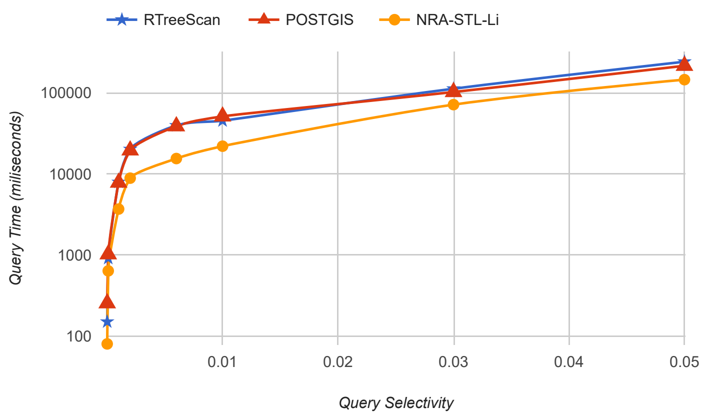


Figure 2.17: Comparing NRA-STL-Li, RTreeScan and POSTGIS for different spatio-temporal query selectivities (100M dataset).

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 "straightforward" approach runs RA-STL-Li or NRA-STL-Li once for each region. Figures 2.19a and 2.19b 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 2.20a and Figure 2.20b 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

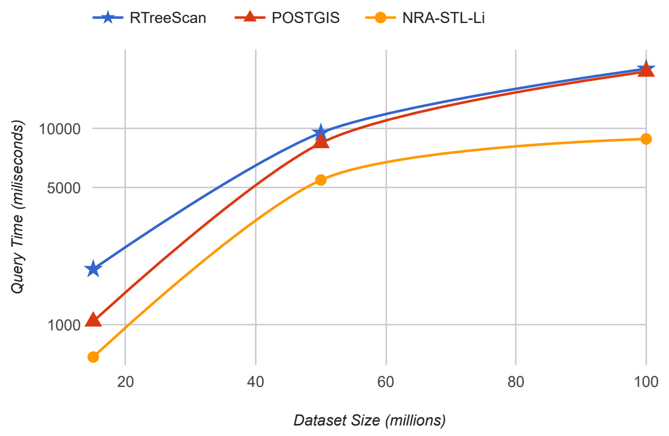
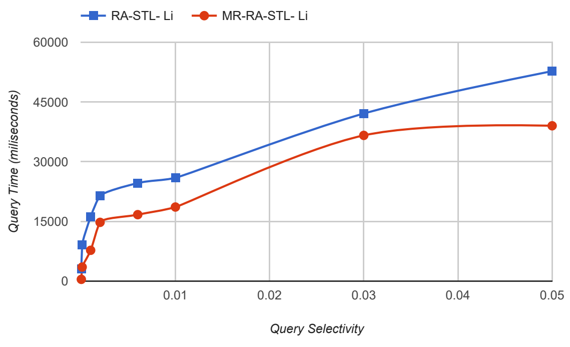


Figure 2.18: Comparing performance of NRA-STL-Li, RTreeScan and POSTGIS for different dataset sizes and selectivity 0.002.

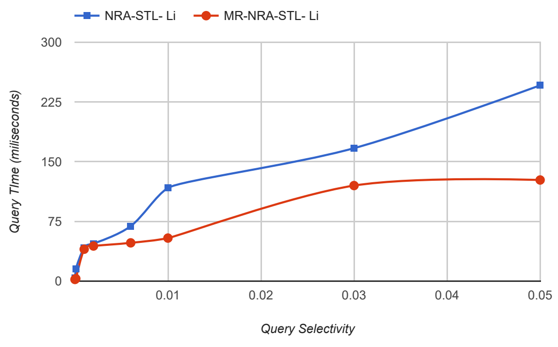
scenario is when there is very large overlap in which case it accesses close to n MBRs (as the R-tree access paths overlap as well). In terms of the number of STLs involved in the query, if a single region query reads l STLs, the multi-region query reads $m * l$ STLs in the worst case and around l in the best case (when the m regions have very large overlap to each other). The exact overhead depends on how far the regions are from each other.

2.7.6 Comparison to Approximate Solutions

We finally compare our approach with the approximate solution of AFIA [65]. 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 2.21 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 $missed$

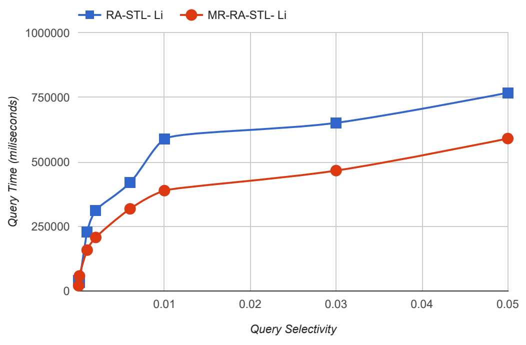


(a) RA-STL-Li vs MR-RA-STL-Li.

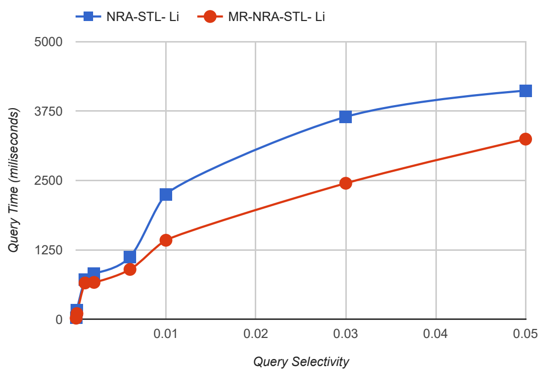


(b) NRA-STL-Li vs MR-NRA-STL-Li.

Figure 2.19: Multi-region query processing using ‘included’ regions.



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



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

Figure 2.20: Multi-region query processing using ‘included’ and ‘excluded’ Regions.

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.

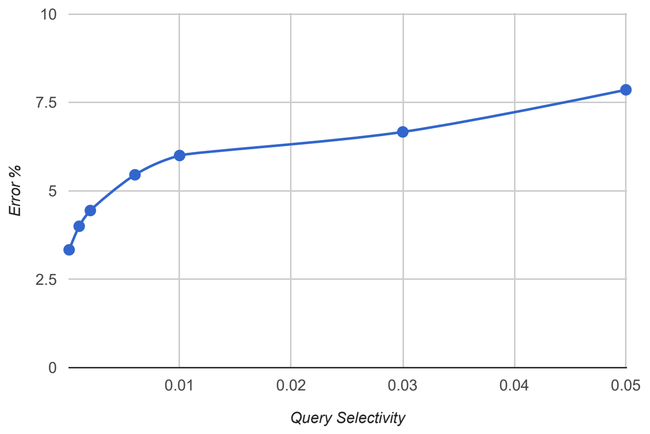


Figure 2.21: Percentage of error for the approximate solution using different selectivities (15M dataset).

2.7.7 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.

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

2.8 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 [16]) so as to process even larger volumes of data. Further, we will study how the proposed indexes can handle high-throughput streaming data.

Chapter 3

Reverse Spatial Top-k Keyword Queries

3.1 Introduction

In this chapter, we focus on the *reverse* problem: given a keyword, we want to find the spatial (or temporal) regions where this keyword is in the top-k most frequent keywords. This query has many applications, and depending on the application different query sizes or time windows are preferable. Consider an advertiser who wants to monitor Twitter posts and identify neighborhoods where a particular product is among the top-k terms discussed. Smaller result areas (say few blocks in size) may be preferable, where electronic billboards can be utilized, to advertise a new product or offer coupons based on the expressed interest in those areas. Location based social media ads can also be instantly purchased. On the other hand, a political candidate's campaign may be interested in identifying larger areas (so that a political rally can be organized) where a specific topic is popular/unpopular. In this application, posts from a wider time window may be considered (the time window is not an explicit parameter in our problem, as it determines the posts collection size; we consider different collection sizes in our experiments). As shown by these examples, in addition to the query term and its importance, the neighborhood size should also be a query parameter.

In this chapter, we investigate such reverse top-k queries on geotagged social posts. Given a user-specified query term q, rank k and a neighborhood size l, the *Reverse Spatial Keyword Query (RSK)* find all the neighborhoods of size l where q is among the k most frequent terms among the posts in those regions.

The problem is challenging because of the large number of possible neighborhoods (which is $O(N^2)$ for N posts). Instead of searching the whole space, we propose an (exact) algorithm that uses a *filtering* step to prune the search space (without missing any answers) and a scan-based *refinement* step to find the answers in the resulting pruned space. We use a grid-based index structure augmented with a materialized sorted term list at each cell to avoid repeated processing of the tweets during query time. To further minimize the RSK query latency, we propose a theoretical model that estimates the optimal grid index cell size. Nevertheless, the refinement step can be slow because of the sheer number of neighborhoods it has to process to find all the answers. Thus we also explore a restricted version of the problem (RSKR) that limits the possible answers to the cells of a query provided grid. In addition to an exact solution, for the RSKR query, we present faster but approximate algorithms where we restrict the number of neighborhood checks using a budget. The proposed algorithms for RSK and RSKR are highly parallelizable. To take advantage of parallelism, we propose a slicing technique that enables distributing the workload of the refinement step among different nodes and thus further reduce the query latency. In summary, our contributions are:

- We introduce the Reverse Spatial Keyword (RSK) query on geo-tagged posts and provide an exact filter and refinement solution. We also consider a restricted version of the query (RSKR) and provide faster exact and approximate algorithms.
- To minimize the RSK and RSKR query latency we propose a theoretical model that finds the optimal index cell size and experimentally evaluate its accuracy using validity tests.
- We explore parallelism for all proposed algorithms, using an efficient load slicing technique to evenly distribute the workload among nodes.
- Using real Twitter datasets, we present a thorough experimental evaluation that verifies our methods' efficiency.

The rest of the chapter is organized as follows: Section 4.2 discusses related work, while Section 4.3 formulates the RSK and RSKR queries. Section 3.4 presents our algorithms for the RSK and RSKR queries. The proposed model to estimate the optimal grid cell size is discussed in Section 3.5. The parallel implementation using the slicing technique appears in Section 3.6. Our algorithms and theoretical models are experimentally evaluated in Section 3.7, while conclusions and future work appear in Section 4.7.

3.2 Related Work

Reverse Spatial Queries: An example in this category is the reverse k-nearest neighbor (RkNN) query which returns all data objects that have the query object in the set of their k-nearest neighbors [9]. Other examples include RkNN for spatial-texual similar objects [49, 50], and the

Reverse top-k Boolean spatial keyword query [32]. While we also look at 'reverse' queries, we return regions instead of data objects.

Vlachou et al. [69] introduced the Reverse top-k query, which, given a"product" p, returns the"weighting vectors" w for which p is in the top-k set. Here, p can be a keyword, while w can be ranges of various types like time interval, spatial region. [69, 70, 72] propose several threshold based algorithms to solve reverse top-k queries, while [59] addresses parallel and distributed processing of the reverse top-k query. Reverse top-k queries can be used to identify the most influential products [71] or monitor the popularity of locations based on user mobility [73].

More related to our work is the reverse *spatial* top-k query, which, given a keyword as input, returns spatial regions based on query-provided preferences like frequency or trend. For example, given a term and a positive integer k, the Reverse Frequent Spatial (RFS) query [29] finds the top k locations on the geographical map where the term is frequent. The key difference is that we can return results of any size while the RFS query returns a list of k cells from the index grid, sorted by the confidence score which is the approximate frequency of the term. GARNET [42] is a system optimized for top-k most trending keyword queries over spatiotemporal streams. As a by-product they also support a restricted version of the proposed RSKR query. We discuss the differences with RSKR in detail in Section 3.7.5, including an experimental comparison.

Density and Burstiness Queries: Related are also works on density-based queries over moving object databases. A spatial area is *dense* if the number of moving objects it contains is above some threshold [36, 57]. While we consider density to identify the result query regions, we are different in that we find regions where a keyword is among the top-k most frequent.

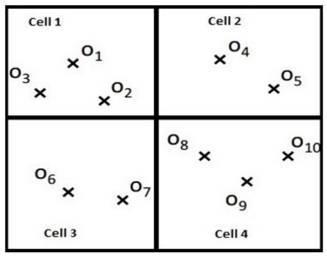
A *burst* is identified when an unusually high frequency (a deviation from the expected frequency) is observed for user provided keyword [44]. [54] examines spatial bursts: given an interval and a term q, identify geographical regions where the observed frequency of q was unusually high, within the interval. In [46], we extended the problem to identifying spatiotemporal regions where a term is bursty. Our work differs in that we provide regions where the term is in the top-k (instead of simply being bursty); also instead of streaming we focus on a disk-based dataset that can be indexed.

3.3 Problem Definition

Let $\mathcal{D} = \{o_1, o_2, ..., o_N\}$ be a dataset with N posts over a spatial rectangle of area A. Each post $o \in \mathcal{D}$ is a tuple $\langle Loc, Terms \rangle$. Here, $o.Loc$ is a spatial point (x, y) that identifies the location of the post and $o.Terms = \{t_1, t_2, ...\}$ denotes the post's terms, where we ignore duplicate terms in the same post. Let $V = \{\cup_{o \in \mathcal{D}}$

$o.Terms\}$ be the vocabulary of all terms. For example, Figure 3.1 shows a collection of 10 posts. The vocabulary, i.e., $\{\cup_{i=1}^{10} o_i.Terms\}$ contains 9 terms. Given a region R, the frequency of term t in R is $f_R(t) = \{count(o_i) | t \in o_i.Terms \ \& \ o_i.Loc \in R\}$. An l**-square neighborhood** is a square region with side length l and sides parallel to longitude and latitude. A query term q is (k, l)-*frequent* at a spatial point p if the frequency of q is among the top-k highest term frequencies in the l-square neighborhood centered at p.

Throughout the chapter, we assume the existence of a grid index I that will facilitate query answering (Figure 3.1a). Each cell in I stores the posts within that cell, sorted along the x-axis. In each cell we also store a *Sorted Term List (STL)*, which is a materialized list of (term, frequency)



Posts	Terms	Posts	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}	$\{t_2, t_6, t_7\}$

Term	Freq
t_2	3
t_4	3
t_1	2
t_3	1
t_6	1

Term	Freq
t_1	3
t_2	1
t_4	1
t_5	1
t_9	1

Figure 3.1: Sample dataset containing 10 posts. (a) the post locations and grid cells, (b) the post terms, (c) STLs for Cell 1 and Cell 3.

pairs, sorted by decreasing frequency. A *pair (t,f)* of the STL indicates that that term t appears in f posts in that grid cell (Figure 3.1c).

Reverse Spatial Keyword (RSK) Query: An RSK query Q is defined by a tuple $\langle k, q, l \rangle$. The answer to RSK Q is the set of spatial regions where q is (k, l)-*frequent* at each point in these regions. Figure 3.2 shows an example of the result regions (deep blue) of an RSK query for the query keyword "york". Any point in the deep blue areas is a center of an l-square neighborhood where q is in the top-K. Note that such result regions can be anywhere (independently of the index cells).

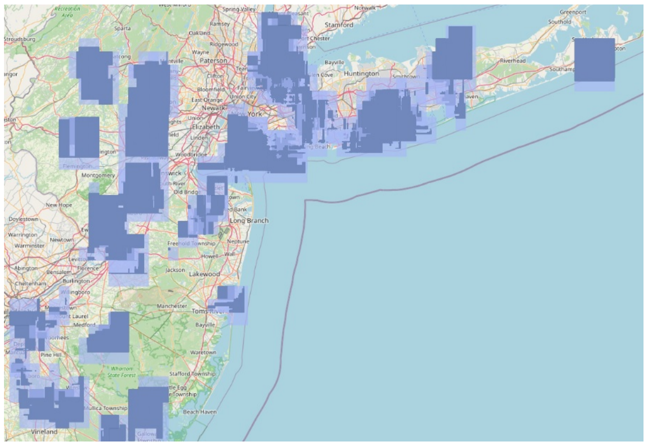


Figure 3.2: RSK (deep blue) and RSKR (light blue) results for keyword "york".

Computing the exact answer to the RSK problem involves checking a large number of neighborhoods and is thus expensive. For that reason, we also propose a resctricted version of the problem (RSKR) defined next.

RSK-Restricted (RSKR) Query: An RSKR query Q_R is again a tuple $\langle k, q, l \rangle$, however, the answer to RSKR Q_R is the set of cells from the index grid I that contain at least one point where q is (k, l)-*frequent*. It is called 'restricted' since the answer is limited among the grid cells. Figure 3.2 shows the result of an RSKR query in light blue (for the same q as above). Note that all the results of RSKR are orthogonal polygons whose sides coincede with the grid but the results of RSK are orthogonal polygons with sides parallel to longitude and latitude. Moreover, from the query definitions, the RSKR result polygons always contain the RSK ones.

3.4 Proposed Algorithms

Consider an RSK query $\langle k, q, l \rangle$. The straightforward algorithm to find all the results for this query needs to scan the whole spatial rectangle A. Unfortunately, the cost of this algorithm is prohibitively expensive for large datasets, as there are $O(N^2)$ different $l \times l$ square windows that must be checked. Assuming that the cost of processing each post is constant, the amount of work for each window is $O(\frac{l^2 N}{A})$, resulting to $O(N^3)$ behavior (since area A and neighborhood size l are not dependent on N).

For both the RSK and RSKR queries, our proposed query processing algorithm consists of two steps: (a) **Filtering step**: Using the stored STLs we identify the cells that are guaranteed to be in the answer (accepts) and the cells that are guaranteed not to be answer (rejects). The rest of the cells are *candidate* cells i.e., the filtering step cannot decide whether they are answers or not.

We process these candidate cells in the refinement step. (b) **Refinement step**: For RSK queries, for each candidate cell, we propose an efficient plane-sweep algorithm to compute the points in that cell where q is (k, l)*-frequent*. For RSKR queries, the refinement step decides, with some confidence, if there is any point in a candidate cell where the query is (k, l)*-frequent* (recall that for RSKR, the answer is returned at the cell granularity). We proceed with the common filtering step; the refinement step for RSK appears in Section 3.4.2 while the (several variants of the) refinement step for RSKR in Section 3.4.3.

3.4.1 Filtering Step

Let l_{min} be the minimum l size that we want to support in the RSK and RSKR queries. Then, the cell size of the grid index I must be $c \leq \frac{l_{min}}{2}$. This condition is necessary for the filtering step of the algorithm to be applicable as we discuss below. Let $l \times l$ be the size of the query neighborhood and $c \times c$ be the size of cells in the grid index. Let $\eta_H = \lceil \frac{l}{2c} \rceil$ and $\eta_L = \lfloor \frac{l}{2c} \rfloor$. We define the *conservative region* $\mathcal{C}_{i,j}$ for a cell $C_{i,j}$ as the union of cells $C_{u,v}$ for which $i - \eta_L < u < i + \eta_L$ and $j - \eta_L < v < j + \eta_L$. Similarly, we define the *expansive region* $E_{i,j}$ for a cell $C_{i,j}$ as the union of cells $C_{u,v}$ for which $i - \eta_H \leq u \leq i + \eta_H$ and $j - \eta_H \leq v \leq j + \eta_H$. Figures 3.3a and 3.3b show the expansive and conservative regions of a cell respectively, using $l = 3c$.

Any point p in cell $C_{i,j}$ is at most $l/2$ distance away from the edges of $\mathcal{C}_{i,j}$ (since $c \leq \frac{l_{min}}{2}$), so p's l-square neighborhood completely contains $\mathcal{C}_{i,j}$. Hence, the $\mathcal{C}_{i,j}$ score is a lower bound for the frequency of the query keyword. Similarly, any point p in cell $C_{i,j}$ is at least $l/2$ distance away from the edges of $E_{i,j}$, so p's l-square neighborhood is completely contained in $E_{i,j}$. Thus the score in $E_{i,j}$ is an upper bound for the frequency of the query keyword. Therefore, if the frequency of the query term q in $\mathcal{C}_{i,j}$ is greater than the frequency of the k^{th} term in $E_{i,j}$, then $C_{i,j}$ is accepted as an answer. This means q is (k,l)-*frequent* for all the points in $C_{i,j}$ and we color the cell as GREEN (accept). On the other hand, if the frequency of q in $E_{i,j}$ is less than the frequency of the k^{th} term in $C_{i,j}$, then there cannot be any point in $C_{i,j}$ that is an answer. Thus q is not (k,l)-*frequent* for any point in $C_{i,j}$ and we color the cell as RED (reject).

The rest of the cells are candidate cells and we color them as GRAY. Only a subset of GRAY cells might be in the answer so they have to go through the refinement step (next section) to calculate which parts of the cell (if any) where q is (k,l)-*frequent*. Figure 3.5 shows an example output of the filtering step.

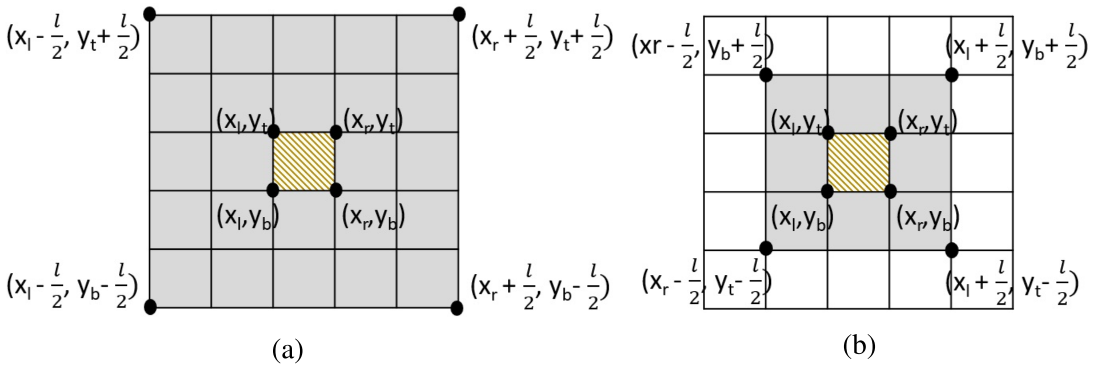

$(x_l - \frac{l}{2}, y_t + \frac{l}{2})$
$(x_r + \frac{l}{2}, y_t + \frac{l}{2})$
(x_l, y_t)
(x_r, y_t)
(x_l, y_b)
(x_r, y_b)
$(x_l - \frac{l}{2}, y_b - \frac{l}{2})$
$(x_r + \frac{l}{2}, y_b - \frac{l}{2})$
(a)
$(xr - \frac{l}{2}, y_b + \frac{l}{2})$
$(x_l + \frac{l}{2}, y_b + \frac{l}{2})$
(x_l, y_t)
(x_r, y_t)
(x_l, y_b)
(x_r, y_b)
$(x_r - \frac{l}{2}, y_t - \frac{l}{2})$
$(x_l + \frac{l}{2}, y_t - \frac{l}{2})$
(b)

Figure 3.3: (a) Cell expansive region for η_H = 2, (b) Cell conservative region for η_L = 1.

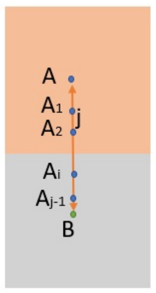


Figure 3.4: Horizontal Jump length estimation.

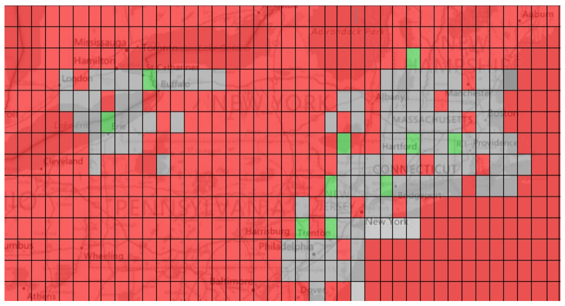
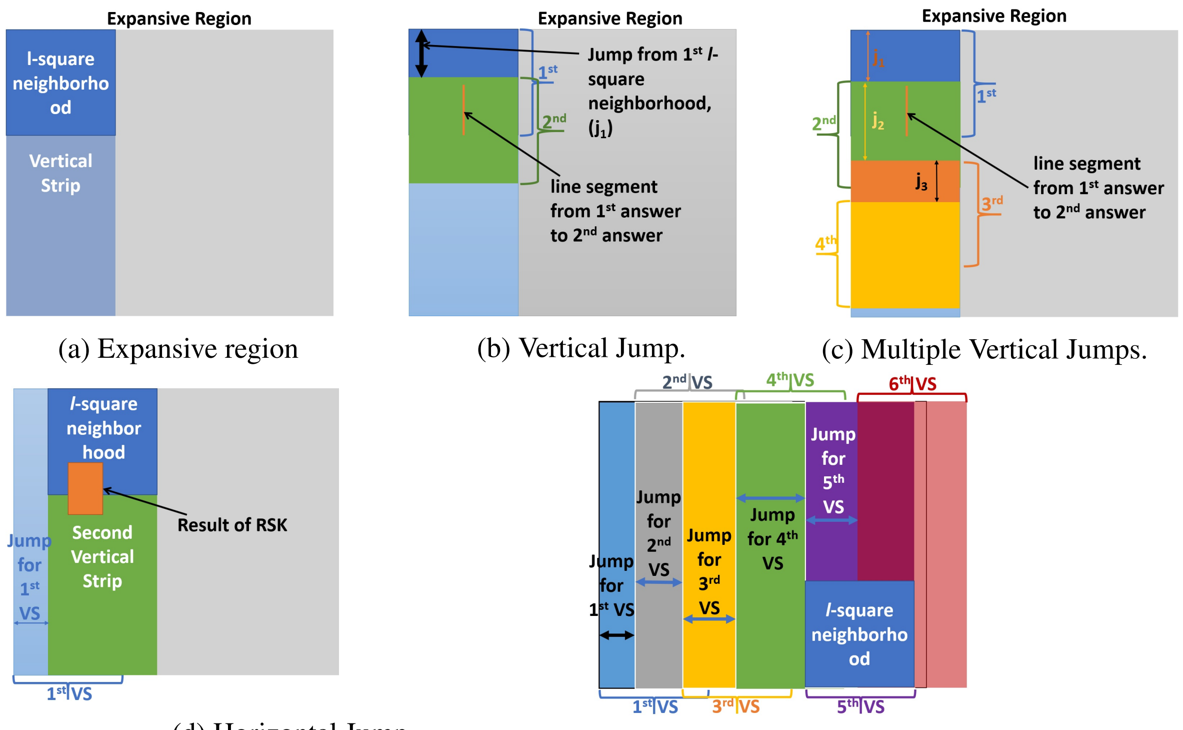


Figure 3.5: Example output of the Filtering step.



(a) Expansive region (b) Vertical Jump. (c) Multiple Vertical Jumps.

(d) Horizontal Jump. (e) Many Horizontal Jumps.

Figure 3.6: Steps of the Proposed Algorithm for RSK query.

3.4.2 Refinement Step for the RSK Query

In this step, we process each candidate cell to find all points in the candidate cell $C_{i,j}$ that are (k,l)-*frequent*. Let candidate cell $C_{i,j}$ have left-bottom corner (x_l, y_b) and right-top corner (x_r, y_t). The expansive region, $E_{i,j}$ of $C_{i,j}$ is a square whose left-bottom corner is $(x_l - l/2, y_b - l/2)$, and right-top corner is $(x_r + l/2, y_t + l/2)$. Clearly, it contains all posts that appear in the l-square neighborhood of any point $p \in C_{i,j}$ (see Figure 3.3a).

We use a plane-sweeping algorithm to identify all the l-square neighborhoods within the expansive region ($E_{i,j}$) of the cell $C_{i,j}$. Throughout the chapter, we discuss one way to traversing the XY-plane by intially fixing the region on X-axis and moving on the Y-axis. After we are done one X-axis, we can move on Y-axis. The order of axes can be easily swapped but it would not change the answer. Before we begin describing the steps of the algorithm, lets define some terms we will frequently use in the chapter.

Vertical Strip (VS): A vertical strip is a rectangle whose top and bottom border equals to that of the expansive region. Its width over X-axis is equal to l. In Figure 3.6a, we see both a vertical strip and an l-square neighborhood with respect to an expansive region. We start from the left side of the expansive region, $E_{i,j}$ (we could have started from the right side as well). We put the vertical slide on the left side of $E_{i,j}$ and the 1^{st} l-square neighborhood, along the top border as shown in Figure 3.6a.

Processing the posts that are contained in the l-square neighborhood and adding them in the hashtable (we call it $termFreqMap$) takes a lot of time. As we already have processed the posts for each cell in the grid, we can leverage STLs to save time. Whenever we are processing an l-square neighborhood, we find out the cells $\mathcal{C}$ that are fully contained in the l-square neighborhood.

Then, we calculate the regions $\mathcal{P}$ that are not covered by the regions covered by the cells in $\mathcal{C}$. Then, we process the posts in $\mathcal{P}$ to calculate the hashtable $termFreqMap$. Finally, we combine all the STLs from the cells in $\mathcal{C}$ into $termFreqMap$. Then we fetch the frequency f_q of q and f_k of the k^{th} most frequent keyword, from $termFreqMap$. We use the QuickSelect algorithm [38] to fetch the score for the k^{th} most frequent term in the *termFreqMap* which takes $O(n)$ time. This allows to avoid the additional cost of sorting all the terms in *termFreqMap* based on their frequencies, to fetch the score of the k-th most frequent term. If $f_q \geq f_k$ then q is (k, l)*-frequent* in the l-square neighborhood, otherwise it's not. In either case, we find the next l-square neighborhood. As we started from the top of the vertical strip, we can only go down along the Y-axis.

Getting the Next l-square neighborhood by Shifting: If we change (add/remove) one post in the l-square neighborhood, we get a new l-square neighborhood. We do this by finding the post (o_1) that is closest to the top border and the post (o_2) that is closest to the bottom border, both on the bottom side. Then, we compute the distance (d_1) between top border and o_1 and the distance (d_2) between the bottom border and o_2. If $d_1 < d_2$, we choose the Y coordinate of 0_1 as the new top border and find our next l-square neighborhood to check. If $d_1 > d_2$, we choose the Y coordinate of 0_2 as the new bottom border and find our next l-square neighborhood to check.

We propose further optimizations by introducing vertical and horizontal jumps. Instead of going post by post to find new windows, we can skip several posts at once. The number of posts that we can safely skip without change of answer is equal to the difference between the worst-case frequency of q, f_w and the k^{th} most frequent term in STL_{C_F} i.e., $|(f_w - f_k)|$. It means that all the l-square neighborhoods that can be created by these skipped posts have the same result as the previous one. Thus the total number of l-square neighborhoods we have to check to find all the answers drastically reduces. We can speed up the refinement process of sweeping along the Y-axis, by using a **Vertical Jump**: instead of shifting by one post, we shift by $|f_q$ - $f_k|$ posts. Since a term is considered present in a post at most once, we are not missing any results. If a vertical jump starts from an l-square neighborhood which is an answer (i.e., q is (k, l)-*frequent*), then it generates a line segment, starting from the center of the current l-square neighborhood to the center of the next l-square neighborhood which is an answer, as shown in Figure 3.6b. The length of these line segments depends on the size of vertical jumps. Any point along this line can be the center of an answer.

Reuse Previous Calculation: Due to the nature of the algorithm, there is much overlap between two consecutive l-square neighborhoods. As in most cases, we jump only a portion of the total posts contained in the query region. As a result, there are many posts that are common between two consecutive l-square neighborhoods. As there is overlap between two consecutive l-square neighborhoods, we can use the calculation of the previous window to assist in the calculation of the next window. To use the calculation of the previous window, we remove the scores of the posts that were part of the previous window but not part of the new l-square neighborhood. Similarly, we add the scores of the posts that are newly added i.e., part of the new l-square neighborhood but not part of the previous l-square neighborhood. We create two separate lists, one containing the removed posts and the other containing the newly added posts. We use these two lists to update the *termFreqMap* from the previous l-square neighborhood to get the *termFreqMap* for the new l-square neighborhood. We continue until the new l-square neighborhood's bottom border reaches or goes beyond the bottom border of the expansive region as shown in Figure 3.6c. It means we have completed the processing of the current vertical strip. Now we start processing of a new vertical strip by shifting on the right side along the X-axis.

After we reach the bottom border of the expansive region or the vertical strip, we have the information for all the l-square neighborhoods that were checked in that vertical strip including their individual jump sizes i.e., $\{j_1, j_2, \ldots j_n\}$. In the lemma below, we show that we can make a safe jump of $j = \frac{j_{min}}{2}$ posts on the X-axis (**Horizontal Jump**) without losing any results, where $j_{min} = \min(j_1, j_2, \ldots j_n)$ is the minimum amount of vertical jump in the vertical strip.

Lemma 1 *: If a vertical strip makes a horizontal jump of $j = \frac{j_{min}}{2}$ posts, where j_{min} was the minimum amount of vertical jump in the vertical strip, we will not miss any result.*

Proof. Intuitively, we are looking for the minimum safe jump radius around any point that can be the center of an l-square neighborhood in the current vertical strip (i.e. the point in along the middle vertical line of the current vertical strip). Let A be the point with the minimum safe vertical jump j_{min}, and B the destination of the jump as shown in Figure 3.4. The point x between A and B (and also in the whole moddle line of the current vertical strip) with minimum safe jump radius is the one where we go with a vertical jump of $\frac{j_{min}}{2}$ from A. x has a remainder safe jump radius of $\frac{j_{min}}{2}$, which is the horizontal safe jump amount we can do. If we would consider another point X' farther from x, then x' would be closer to B (and hence its horizontal safe jump would be bounded by $\frac{j_B}{z}$, where z would be smaller than 2 (closer to B than the midpoint x) and $j_B \geq j_{min}$, so $\frac{j_B}{z} \geq \frac{j_{min}}{2}$. ■

Figure 3.6d shows an example of horizontal jump. Next, we process the new vertical strip in the same way as mentioned above. We keep shifting vertical strips, until the right border of the new vertical strip reaches or goes beyond the right border of the expansive region, $E_{i,j}$.

This concludes the processing of one candidate cell. The overall algorithm is formally presented in Algorithm 8. Figure 3.6e shows an example where the 6^{th} vertical strip is beyond the expansive region boundary (shown as a purple vertical line on the right), so the refinement step stops at the 5^{th} vertical strip. Each horizontal jump stretches the line segment(s) generated in one vertical strip into rectangles (as shown in Figure 3.6d). The width of these rectangles depends on the horizontal jump size. The orthogonal polygons in the RSK query result (Figure 3.2) are created by the union of all these rectangles. Note that q is (k,l)-*frequent* at any point within these polygons.

Using vertical and horizontal jumps, the total number of l-square neighborhoods that the RSK algorithm checks is $\frac{N^2}{j}$ (where j is the average jump size), which results to $O(\frac{l^2N^3}{jA})$ running time. This is still $O(N^3)$ however, j is a large constant resulting in much better performance in practice than the straightforward algorithm.

3.4.3 Refinement Step for the RSKR Query

The refinement step of the RSK query algorithm checks a large number of l-square neighborhoods which leads to high query latency. One approach to lower the query latency is to stop processing within a candidate cell as soon as the first l-square neighborhood where q is (k,l)-$frequent$ is found in that cell. Thus the refinement step of the RSKR query algorithm returns exactly those cells that have answers. But for the candidate cells where there is no result, this simple approach will still check all windows centered within this cell (i.e. the same approach as the exact solution).

Coordinate Division: To reduce query latency, we propose an approach that divides the search space and checks a bounded number of l-square neighborhoods per candidate cell, based on a technique we call *Coordinate Division (CD)*. This technique is applied on each candidate cell in

Algorithm 8 $RSK(cell, q)$

```
Require: Query term $q$ and cell contains a STL for the posts in that CELL
Ensure: Return all the $l \times l$ sized squares where $q$ is among the top-k
 1: $posts \leftarrow getAdjacentposts(c)$
 2: sortpostsLongitude(posts)
 3: while $true$ do
 4:    $cell \leftarrow Cell(left, topBorder)$
 5:    $currentPosts \leftarrow postsIn(posts, cell)$
 6:    $sortpostsLatitude(currentPosts)$
 7:    $top \leftarrow topBorder$
 8:    while $true$ do
 9:       $cellY \leftarrow newCell(VerticalStrip, left, top)$
10:       for each post in currentposts do
11:          $currentPostsY \leftarrow post$
12:          if not in previousGrid then
13:             $newlyAddedpost \leftarrow post$
14:       if $prevGridexists$ then
15:          $termFreqMap \leftarrow processRemovedposts()$
16:          $termFreqMap \leftarrow processNewlyAddedposts()$
17:       else
18:          $termFreqMap \leftarrow processposts(currentPostsY)$
19:       $q_{score} \leftarrow termFrequencyMap.get(q)$
20:       $k^{th}Score \leftarrow quickSelect(termFreqMap.values(), k)$
21:       if $q_{score} \geq k^{th}Score$ then
22:          $result \leftarrow cellY$
23:          $c.color \leftarrow GREEN$
24:       $jump \leftarrow |k^{th}Score$ - $q_{score}|$
25:       $jumps \leftarrow add(jump)$
26:       $prevGrid \leftarrow cellY$
27:       if $cell.bottom \geq bottomBorder$ then
28:          break
29:    $minJump \leftarrow min(jumps)$
30:    $VerticalStrip \leftarrow jump(VerticalStrip, minJump)$
31:    if $cell.right \geq rightBorder$ then
32:       break
33: return $result$
```

iterations. In one iteration, within each candidate cell, a random point is chosen and used as the center of the l-square neighborhood. If that neighborhood is an answer the algorithm stops processing that cell. If not, the chosen point is used to divide the cell space into four regions. A random point is then chosen in each of the four regions. If an answer is found in any of the four l-square neighborhoods centered on these points, the algorithm stops processing this cell. Otherwise, a *diff_value* is calculated for each of the four l-square neighborhoods; this diff_value is the difference between the score of the query term q and the score of the k^{th} most frequent term in the l-square neighborhood. The algorithm picks the region with the point that has the lowest diff_value and continues by dividing that region into four parts as before. This iteration stops either when a result is found or an upper bound for the number of divisions is reached. Checking whether an l-square neighborhood is an answer or not is similar as with the RSK query algorithm with one variation. Since we randomly choose points, the algorithm will use the previous $termFreqMap$ only when there is enough overlap (at least 50%) between subsequent l-square neighborhoods. Note that because points are picked randomly, a CD iteration over a candidate cell may miss some results. Hence, we allow the RSKR query algorithm to run multiple iterations on candidate cells where no answer is found.

There are thus two parameters affecting the RSKR query algorithm performance: (i) the number of divisions, and (ii) the number of iterations. Increasing any of these parameters improves the accuracy at the expense of query latency. The overall algorithm is formally presented in Algorithm 9.

We implemented two additonal heuristics on the RSKR query algorithm: (i) **Partial STL**: When a cell is partially contained in the l-square neighborhood, instead of identifying and processing the posts that are contained in this l-square neighborhood, we access the STL of that cell and

Algorithm 9 $RSKR - Approximate(GRID, q)$

Require: Query term q and GRIDINDEX is the space covered divide into cells. Each CELL in GRIDINDEX contains a STL for the posts in that CELL

Ensure: Color all the cells in the GRIDINDEX to indicate whether there is any $l \times l$ sized squares in the cell where q is among the top-k

```
1: while there is more randomRestart do
2:   while true do
3:     $randomPoint \leftarrow calculateRandomPoint(cell)$
4:     $topLeftCell \leftarrow calculateTopLeftCell(cell)$
5:     $scoreTopLeft \leftarrow calculateScore(topLeftCell)$
6:     if $scoreTopLeft > 0$ then
7:       $cell.color \leftarrow GREEN$
8:       break
9:     $topRightCell \leftarrow calculateTopRightCell(cell)$
10:    $scoreTopRight \leftarrow calculateScore(topRightCell)$
11:    if $scoreTopRight > 0$ then
12:      $cell.color \leftarrow GREEN$
13:      break
14:    $bottomLeftCell \leftarrow calculateBottomLeftCell(cell)$
15:    $scoreBottomLeft \leftarrow calculateScore(bottomLeftCell)$
16:    if $scoreBottomLeft > 0$ then
17:      $cell.color \leftarrow GREEN$
18:      break
19:    $bottomRightCell \leftarrow calculateBottomRightCell(cell)$
20:    $scoreBottomRight \leftarrow calculateScore(bottomRightCell)$
21:    if $scoreBottomRight > 0$ then
22:      $cell.color \leftarrow GREEN$
23:      break
24:    $max \leftarrow max(scoreTL, scoreTR, scoreBL, scoreBR)$
25:    if $max \leq score$ then
26:      break
27:    if $max == scoreForTopLeft$ then
28:      $rightBorder \leftarrow randomX$
29:      $bottomBorder \leftarrow randomY$
30:    if $max == scoreForTopRight$ then
31:      $leftBorder \leftarrow randomX$
32:      $bottomBorder \leftarrow randomY$
33:    if $max == scoreForBottomLeft$ then
34:      $rightBorder \leftarrow randomX$
35:      $topBorder \leftarrow randomY$
36:    if $max == scoreForBottomRight$ then
37:      $leftBorder \leftarrow randomX$
38:      $topBorder \leftarrow randomY$
39:    else
40:      break
41: return $result$
```

multiply its scores by the percentage of overlap. (ii) **STLOnly**: We can further speed up latency by only considering the STLs of the cells that are fully contained in the l-square neighborhood. The effects of these heuristics on query latency and accuracy are examined in the experimental section. The RSKR results are produced by the union of the returned grid cells; hence they are orthogonal polygons aligned to the grid (shown in light blue in Figure 3.2).

3.5 Optimal Cell Size Estimation

This section presents a theoretical analysis for the processing cost of the RSK and RSKR queries. The objective is to find the optimal cell size that minimizes the processing cost of the two corresponding refinement steps presented in Sections 3.4.2 and 3.4.3. First, we discuss estimating the optimal cell size for RSKR which is necessarily the calculation we need to estimate optimal cell size for a single l-square neighborhood. After that, we will use the calculation to estimate the optimal cell size for RSK problem. Table 3.1 summarizes the notation used in the analysis.

3.5.1 Analysis of the RSKR refinement step

Let, N be the total number of posts in the input dataset, c be the side length of the square cells, l be the side length of the square l-square neighborhood, and A be the total area of the minimum bounding rectangle (MBR) that covers the input dataset. The area of one cell is c^2, area of the l-square neighborhood is l^2 and the total number of cells is $\frac{A}{c^2}$. Assuming a uniform distribution of the data points, the average number of posts per cell is $\rho = \frac{Nc^2}{A}$. Let the number of cells that are fully contained and partially contained in the l-square neighborhood be I and P, respectively. The

Symbol	Description
N	Total number of posts in the input
A	Total area covered by the dataset
c	Side length of the square cell
l	Side length of the square l-square neighborhood
ρ	Average number of posts per cell
y	Average number of terms per post
K	First parameter in Heap's Law
β	Second parameter in Heap's Law
I	Number of cells fully contained in the l-square neighborhood
P	Number of partial cells in the l-square neighborhood
N_C	Number of posts in a cell
N_{VS}	Number of posts in a vertical strip
N_E	Number of posts in an expansive region

Table 3.1: Notations used throughout Theoretical Analysis.

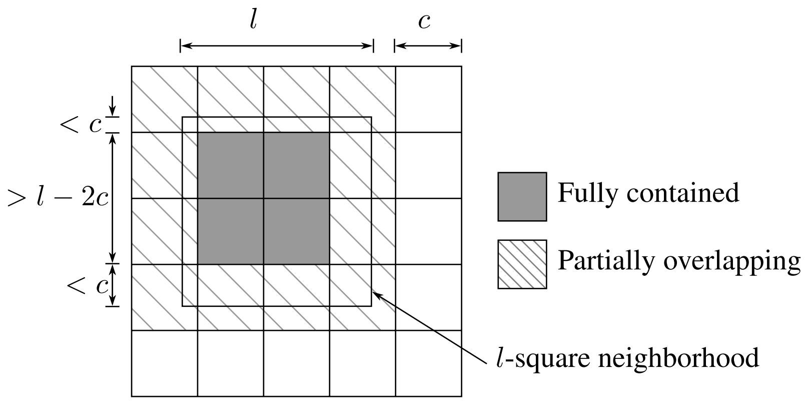


Figure 3.7: Fully contained cells and paritally intersecting cells in the l-square neighborhood.

total cost of processing an l-square neighborhood is divided into two parts, the cost of processing fully contained cells and the cost of the partially contained cells.

Cost of processing fully contained cells: To compute the cost of processing fully contained cells, we compute the total number of fully contained cells I and multiply this by the average cost of processing one cell.

Lemma 2 *There is at least $(\frac{l}{c} - 1)^2$ cells that are fully contained in the l-square neighborhood.*

Proof. As illustrated in Figure 3.7, there is at most two partially overlapping cells along each dimension, i.e., one partial cell on each end. The length of overlap between the query region and a partial cell is less than c. This means that the length of all fully contained cells is $> l - 2c$. Along that length, the number of cells is larger than $\lceil l/c \rceil - 2$ cells. Since the number of cells is an integer, the number cells along one dimension is at least $\lceil l/c \rceil - 1$. This means that there is at least $(l/c - 1)^2$ fully contained cells. ■

From Lemma 2, the total number of fully contained cells is $I = (\frac{l}{c} - 1)^2$. Next, we will calculate the average cost of processing one fully contained cell.

The RSKR refinement step processes fully contained cells by simply merging their STLs into one. Each STL contains a list of term frequencies. First, we need to compute the average size of one STL, i.e., the number of unique terms in one cell. To estimate the number of unique terms in one cell we use Heap's Law [26], $STL_{size} = K(\rho \cdot y)^{\beta}$, where $\rho = \frac{c^2 N}{A}$ is the total number of posts in a cell, y is the average number of words per post, K and β are two free parameters of Heap's Law that are calculated once for the entire dataset. The total processing cost of all fully contained cells, $T_I = I \cdot STL_{size} = (\frac{l}{c} - 1)^2 \cdot K(\rho \cdot y)^{\beta} = (\frac{l}{c} - 1)^2 \cdot K(\frac{c^2 N y}{A})^{\beta}$.

Cost of processing partially intersected cells: We compute the cost of processing partially intersected cells by splitting it into two steps, *fetching* and *processing*. The fetching step scans all posts inside partially intersected cells to find the posts that are inside the l-square neighborhood. The processing step scans all the terms in all the fetched posts to update the overall term frequencies in the l-square neighborhood. The details are provided below.

Lemma 3 *There is at most $4\frac{l}{c}$ partially intersecting cells in the l-square neighborhood..*

Proof. According to Lemma 2, there is at least $\frac{l}{c} - 1$ fully contained cells along each of the two dimensions. Additionally, there is at most two partially intersecting cells on each end of these cells. This makes the total number of partially intersecting cells that surround the fully contained cells from the four directions $4(\frac{l}{c} - 1)$. In addition, there are four additional partially intersecting cells on the four corners. This makes the total number of partially intersecting cells $4\frac{l}{c}$. ∎

According to Lemma 3, there are $4\frac{l}{c}$ cells intersecting with the l-square neighborhood. Since these cells are not fully contained in the l-square neighborhood, we cannot simply use their STLs and we will need to fetch and process the individual posts inside the l-square neighborhood. Assuming no index inside each cell and a unit cost of processing each post, the cost of fetching the posts is equal to the total number of posts in partially contained cells which is $4\frac{l}{c} \cdot \frac{c^2 N}{A} = \frac{4lcN}{A}$.

Second, the cost of *processing* the posts is equal to the total number of terms (not unique terms) in all posts inside the l-square neighborhood. The area of l-square neighborhood is l^2, the area covered by the fully contained cells is $(\frac{l}{c} - 1)^2 * c^2$, thus, the area covered by partial cells is $l^2 - (\frac{l}{c} - 1)^2 * c^2 = 2lc - c^2$. Assuming uniform distribution and y terms per post, the total cost of processing all partially intersecting cells, $T_P = \frac{4lcN}{A} + \frac{yN}{A}(2lc - c^2)$. The total cost for processing a single l-square neighborhood, θ is shown in Equation 3.1.

$$\theta(c) = (\frac{l}{c} - 1)^2 * K(\frac{c^2 yN}{A})^{\beta} + \frac{4lcN}{A} + \frac{yN}{A}(2lc - c^2) \tag{3.1}$$

The optimal cell c_* is the cell size that minimizes the value of θ in Equation 3.1; to find c_* we use Wolfram Alpha [76].

If the filtering step generates a total of G number of candidate (gray) cells for a query keyword, in the worst case, the refinement step of RSKR will check $\frac{B}{G} * \theta(c)$ l-square neighborhoods, where B is the total budget allocated for a query keyword which is equally divided among the candidate cells. With the change in the number of candidate cells, the budget per candidate cell changes as well.

Example: Let N = 15 million, y = 3, l = 1, A = 220, K = 1.92 and β = 0.07197. Equation 3.1 has a local minimum at $c = 0.0346488$ and local maximum at $c = 0.840214$.

3.5.2 Analysis of the RSK refinement step

We use the same notations used in the previous section to analyze the running time of the exact RSK refinement algorithm. We break down the running time of the RSK algorithm as follows: **1.** All the posts in the expansive region are sorted by x. **2.** The posts in the first vertical strip are sorted by y. **3.** The first l-square neighborhood is processed as analyzed in Section 3.5.1. **4.** Subsequent l-square neighborhoods in each vertical strip are processed through vertical jump. **5.** The next vertical strip is identified and steps 2-4 are repeated until all vertical strips are processed.

In the next part, we analyze the processing cost for the above steps in order.

1. To estimate the sorting cost, we need to estimate the total number of posts in the expansive region. The number of cells in the expansive region is $(2 \cdot \frac{l}{2c} + 1)^2$ = $(\frac{l}{c} + 1)^2$. Assuming uniform distribution of posts over the space, post count in expansive region, $N_E = \left(\frac{l}{c} + 1\right)^2 \cdot \frac{c^2 N}{A}$. So cost of sorting posts in expansive region, $sort_E$ = $N_E \log(N_E)$.

2. Similarly, the area of the vertical strip is $l \cdot c \cdot (\frac{l}{c} + 1)$. Assuming uniform distribution, post count in vertical strip is: $sort_{VS} = l \cdot c \cdot \left(\frac{l}{c} + 1\right) \cdot \frac{N}{A} \log(N_{VS})$.

3. The first l-square neighborhood in a vertical strip is processed similar to RSKR query in Section 3.5.1 and the processing cost is given by Equation 3.1.

4. To compute the cost of subsequent l-square neighborhoods, we need to estimate the size of the vertical jump j. We estimate the jump size to be the absolute difference between the estimated frequencies of the query keyword and the k^{th} keyword, $|f_q - f_k|$ in the l-square neighborhood, $j = \left|\frac{l^2c^2yN}{A} \cdot \left(\frac{1}{r_q} - \frac{1}{k}\right)\right|$, Where r_q is the rank of the query keyword in the dataset. Given the jump size, we can estimate the number of l-square neighborhoods checked per vertical strip, $l_{Count_{VS}} = \left(N_{VS} - \frac{N \cdot l^2}{A}\right)/j$. From steps 2-4, the cost of processing one vertical strip is, $cost_{VS} = sort_{VS} + \theta + j \cdot y \cdot l_{Count_{VS}}$.

5. To estimate the number of vertical strips, VS_{Count}, we compute the size of the horizontal jump to be half that of the vertical jump, i.e., $j/2$. Estimated number of vertical strips is, $VS_{Count_E} = \left\lceil \frac{2 \cdot (N_E - N_{VS})}{j} \right\rceil$.

To sum it up, step 1 is performed only once, steps 2-4 are performed for each vertical strip, and step 5 determines the number of vertical strips. Therefore, the total cost of the RSK refinement step is as follows $\theta_F(c) = sort_E + cost_{VS} = sort_E + (sort_{VS} + \theta + jy \cdot l_{Count_{VS}}) \cdot VS_{Count_E} = N_E \log(N_E) + (N_{VS} \log(N_{VS}) + \theta + jy \cdot l_{Count_{VS}}) \cdot VS_{Count_E}$. The optimal cell c_* is the cell size that minimizes $\theta_F(c)$ as before, we find c_* using Wolfram Alpha [76].

3.6 Parallel Implementation

RSK parallelism: As the experimental results will show, the RSK query provides an exact answer, but it is time consuming. Among the two parts of the algorithm, filtering is very fast because it only processes the STLs (i.e. not the actual posts). In contrast, the refinement step processes the actual posts, hence it takes much longer (about 99% of the query latency). In this section, we explore how parallelism can be used to further improve query latency for the refinement step. Our aim is to divide the work among the participating nodes so as to achieve balanced load. During the RSK refinement step, the time taken by each candidate cell varies substantially (see Figure 3.8). Since the RSK algorithm as presented, checks l-square neighborhoods first along the Y-axis, we introduce a vertical *slicing* mechanism that divides the work from busy candidate cells into smaller independent units (slices) that can be processed in parallel.

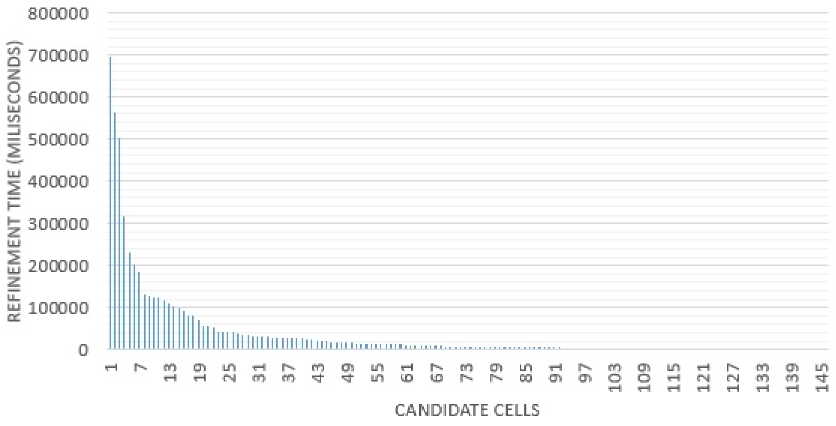
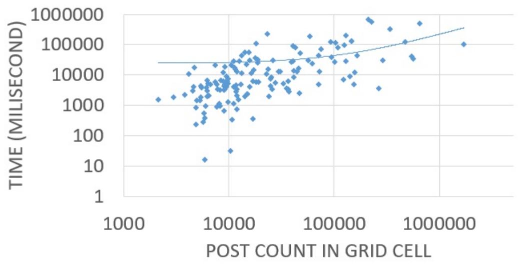


Figure 3.8: Refinement time for different candidate cells for q = ”home” and k = 5.



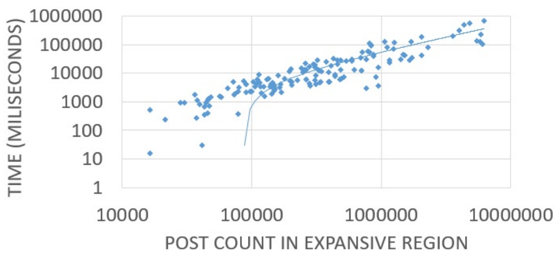


Figure 3.9: Correlation between refinement time and cell post count (left), expansive region post count (right), for q = “home”, k = 5.

Indicators of Large Refinement Time: Since the actual refinement time for a particular candidate cell is query dependent and not known until the refinement step is executed, we would like to find indicators that can accurately estimate the refinement time and are either precomputed (i.e.

query independent), or easy to compute at query time. For each candidate cell we considered four such indicators, namely: (a) *its post count*, (b) *the post count in the expansive region of the cell*, (c) *its jump size* and (d) *the jump size in its expansive region*. Among them, (a) is query independent, while the rest are easy to compute after the filtering step. For example, (c) uses the jump size ($|f_q - f_k|$), while (d) depends on the jump size and l. Similarly, (b) can be easily computed per candidate cell using l and cell post counts.

Our experiments showed that the jump size indicators are also dependent on the post count (which is to be expected because the maximum possible jump is equal to the post count). Hence we concentrate on the post count based indicators; Figure 3.9 plots the correlation between the refinement time and the post counts (cell or expansive region). Clearly, the *post count in the expansive region* (termed as N_E) is the indicator of choice as it has the highest correlation with refinement time. Intuitively, the posts in the expansive region are a better representative of the posts needed to answer the RSK query for a given candidate cell.

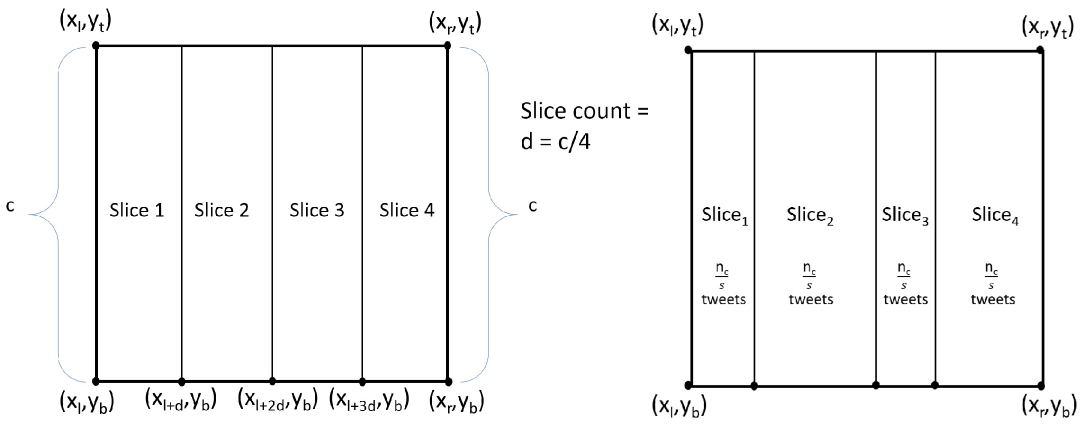


Figure 3.10: Equal-width (left) and equal-post slicing (right).

Slicing: Let G be the set of the candidate cells provided by a query's filtering step. Below we discuss a heuristic that slices candidate cells with high N_E. In particular, each candidate cell is assigned a slice count s (the number of slices for that cell).

Let $\mathcal{M}$ be the number of cores in the cluster. Summing N_E for all cells in G represents the amount of work (all posts that need to be considered) needed by the RSK refinement step. Ideally, we would like to divide this workload equally among all nodes, but this is not possible since each N_E is of different size. To enable easier distribution we divide the workload into $\gamma * \mathcal{M}$ slices where γ is a constant which indicates the average number of slices per node. By varying γ, we can control the total number of slices created. The average work per slice is $\tau = \frac{\sum_G N_E}{\mathcal{M} * \gamma}$ and the slice count for a given candidate cell is $s = \lfloor \frac{N_E}{\tau} \rfloor$. The proposed heuristic assigns s slices to the candidate cells with $s > 1$.

Figure 3.10 (left) shows a cell sliced vertically into four equal-width slices ($s = 4$). While processing a slice in the refinement step, we use the boundary of the slice instead of the boundary of the cell. For example, processing the third slice only checks for the l-square neighborhoods whose centers are in the rectangle with left-bottom corner (x_{l+2d}, y_b) and right-top corner (x_{l+3d}, y_t) (where $d = \frac{c}{s}$). Since posts may not be uniformly distributed within a cell, we also explored equal-post slicing. Figure 3.10 (right) shows an example. Here, the vertical strips are positioned on the X-axis so that each slice has the similar number of posts. Our experiments (not shown) showed that using equal-post slicing offered 8-15% improvement in query latency.

Processing a particular slice needs to be independent; hence the node that is assigned a given slice should have all the data needed for processing the refinement step on this slice. Ideally, one could identify the posts and STLs for the expansive region of each slice. Since this is time-consuming, we instead send to the assigned node the posts and STLs included in the expansive region of the parent cell. We call this data the Cell Data Store (CDS). All slices of a given cell get the same CDS.

The final step assigns the slices (and their CDS) to the $\mathcal{M}$ cluster nodes. All slices are stored in a sorted list (in decreasing order) according to their parent cell's N_E. We start with $\mathcal{M}$ empty buckets (one bucket for each cluster node) and assign the top $\mathcal{M}$ slices from the list sequentially to the buckets. Buckets are then added in a priority queue that orders them (in decreasing order) according to their aggregate N_E. The bucket with the smallest aggregate N_E is assigned the next slice from the sorted list; this process continues until all slices are assigned.

We presented slicing in a vertical way since it has to obey the same direction as the plane sweep algorithm. If instead we had used a horizontal sweep line, slicing would be horizontal. However, we have found that performing slicing in both directions would not be as effective. This is because slicing in both dimensions will reduce the number of vertical jumps. Further, the size of a horizontal jump is equal to half of the minimum of all the vertical jump sizes ($\frac{jump_{min}}{2}$), and thus is typically smaller than a vertical jump.

RSKR parallelism: We also explore parallelism for the RSKR query algorithm. Since this algorithm terminates as soon as a result is found while processing a given cell, slicing will not help. Since slices of a cell are processed at different nodes, if a result is found on a slice we would need to terminate all other slices of the same cell. Instead, to parallelize RSKR, we distribute the candidate (gray) cells among nodes using a bucket-based approach as above (so that every node gets a similar amount of tweets).

3.7 Experiments

3.7.1 Setup

Hardware: We experimentally evaluate the presented algorithms, both for single-node and multi-node environments. All single-node experiments are run on a machine featuring an Intel Core i-7 processor (8 cores) with 16 GB of RAM and 7200 rpm hard drive. The single-node experiments use all available (eight) cores on the processor. All multi-node experiments are run on an AWS Spark cluster. Each slave machine was *r5.xlarge* with 4 vCPUs and 32GB of memory. We run one executor per core (vCPU).

Datasets: Using the Twitter streaming API [5], we collected all geo-tagged (i.e., tweets that have the user's GPS location or the user's 'Twitter Place'), English-based tweets (i.e., excluding empty tweets or tweets with only URLs) from a rectangle that contains the New York state and surrounding areas (i.e., region A has GPS coordinates: -91, -66, 46, 36) for the six month period from August 2014 to January 2015. This resulted in a dataset with 15M geo-tagged tweets (12GB in size). In particular, since most users do not typically reveal their phone's GPS [22, 41], there were only about 3% tweets with actual GPS location. For the rest of the geo-tagged tweets (i.e., those with the user's 'Twitter Place') we used as location, a random point in the provided polygon that the Twitter API shares for 'Place'. Each tweet record has the tweet's spatio-temporal coordinates and its terms. After removing stopwords, each tweet has an average of 5 terms.

As discussed in Section 3.4, the RSK query is computationally very expensive with respect to the number of tweets involved in the query, which is application dependent. This number can increase either by collecting (over time) more tweets over an area, or by increasing the area's size. Our experiments use datasets derived from the full dataset above (by keeping the area fixed

Symbol	**Rank**	**Keyword**
Q_1	1	Love
Q_5	5	Good
Q_{10}	10	Sorry
Q_{25}	25	Home
Q_{50}	50	Hope
Q_{200}	200	Change

Table 3.2: Keyword ranks.

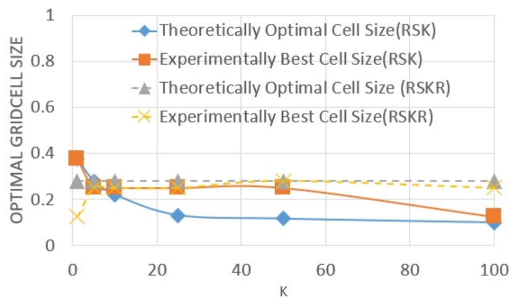
but limiting the time interval), with sizes varying from 10K tweets to the full dataset of 15M tweets. This is needed for testing the scalability of our algorithms as well for emulating scenarios where an application deals with an area/interval that contains fewer data. For comparison purposes, today's Twitter API provides around 28.5K tweets with GPS location per day for the above rectangle A.

Query keywords: In our experiments, we use query keywords that have different ranks in the dataset based on their frequencies. Table 3.2 shows six keywords with ranks 1, 5, 10, 25, 50 and 200 based on their frequency in the full 15M dataset. Depending on the dataset used in each experiment, the actual keyword at a specific rank may be different. Furthermore, throughout this section, if not otherwise specified, we use k = 10.

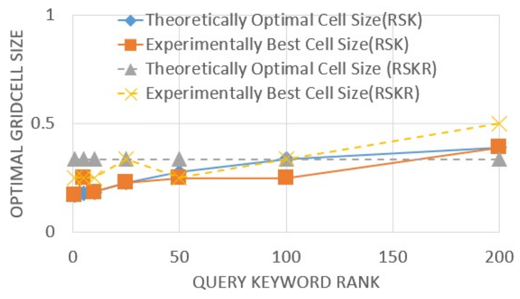
Index Structure: We divided the space covered by each dataset into a uniform grid of square cells with each cell containing tweets that are in the geographical area covered by that cell. Each cell is enhanced with a STL of it's tweets.

3.7.2 Model Validation

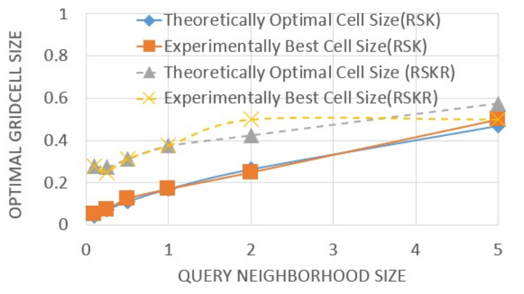
To check the optimal cell size estimator model, we run stability and validity tests. In the stability tests, we depict how the optimal cell size produced by the model differs from the experimentally obtained 'best' cell size. In particular, we experimented with a fixed set of square cell sizes, with side length: 0.5, 0.25, 0.125, 0.1, 0.05, 0.025, 0.02, 0.01, 0.005 (degrees of longitude and latitude). In each experiment we report the *experimentally best cell size* as the cell size that showed the smallest query latency.



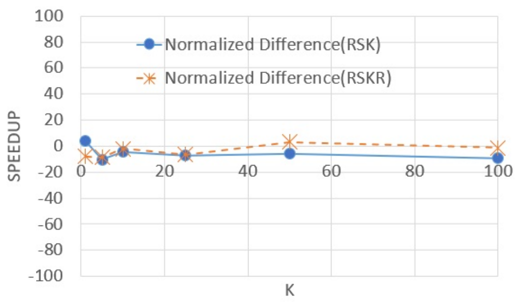
(a) Validity test (cell size) - varying k.



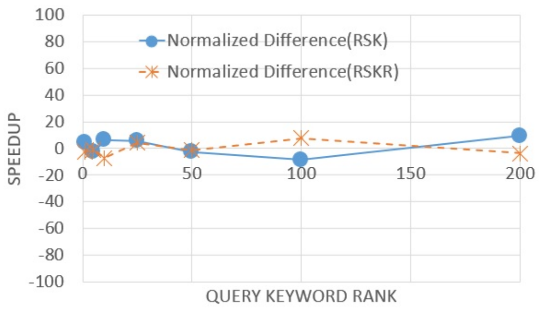
(b) Validity test (cell size) - varying query keywords.



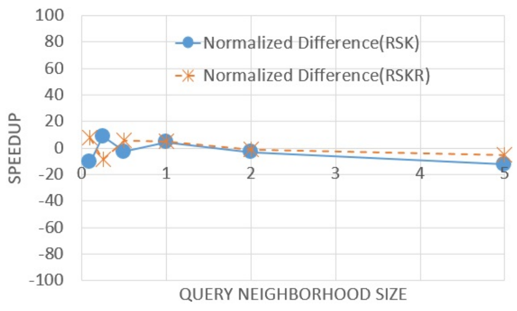
(c) Validity test (cell size)- varying query sidelength.



(d) Validity test (speedup) - varying k.



(e) Validity test (speedup)- varying query keywords.



(f) Validity test (speedup)- varying query sidelength.

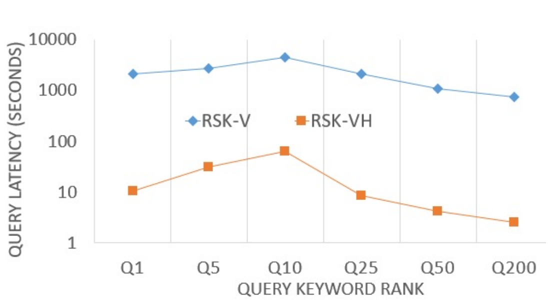
Figure 3.11: Validity test with respect to cell sizes and speedup for cell size estimator.

The validity results with repect to cell size for the RSK and RSKR problem appear in Figures 3.11(a)-(c). In these experiments, we used the RSK algorithm with vertical and horizontal jumps (Algorithm 8) on the full dataset (15M) and varied k (from 1 to 100), the query keyword rank (Q_1 through Q_{200}) and the query neighborhood size l (from 0.1 to 5 degrees). As it can be seen, the theoretically optimal cell size for RSK and the experimentally 'best' are very close in all experiments, while varying different parameters. We also observe that with the change of parameters i.e., k, Q and keywords, the optimal cell size remains stable and doesn't show any abrupt changes. Figures 3.11(d)-(f) present the validity tests with respect to speedup for the RSK problem using the same dataset and varying the same parameters. Here the normalized difference between the query latencies, for the theoretically optimal cell size c_θ and the experimentally 'best' c_{exp} is depicted, as the speedup: $\frac{Latency(c_\theta)-Latency(c_{exp})}{Latency(c_\theta)}$. As it can be observed from these figures, the difference between the query latencies remains low (within 10%). We also run stability and validity tests for the RSKR problem using Algorithm 9 on the large dataset (15M) varying the same parameters. As before, we observe that the model is quite accurate.

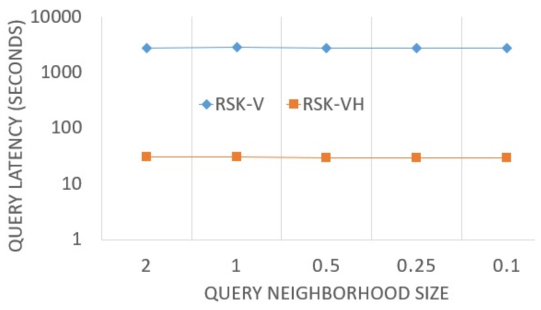
For simplicity, in the rest of the chapter we fix the value of k to 10; for this k the theoretically optimal cell size c was close to 0.25 which is the cell size we used in the following experiments (unless otherwise mentioned).

3.7.3 Single Node Evaluation

We first examine the performance of the **RSK query**. For these experiments, we use a dataset of 10K tweets taken by random sampling from the full dataset. We compare three algorithms, namely, the baseline (**RSK-W**, i.e. without jump), RSK with only vertical jump (**RSK-V**) and RSK with both vertical and horizontal jump (**RSK-VH**, described in Algorithm 8). Figures 3.12a and 3.12b present the query latencies of the algorithms while varying the query keyword positions q (using $l = 0.5$) and the query neighborhood size l (using keyword Q_5) respectively. The baseline approach



(a) Varying query keyword



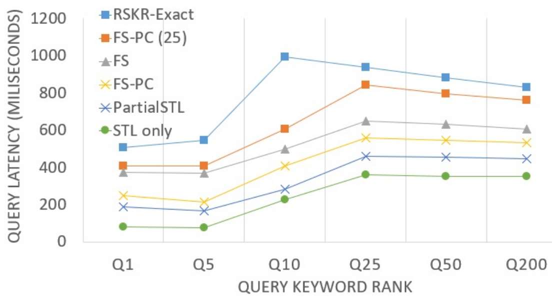
(b) Varying neighborhood sizes

Figure 3.12: RSK query latency.

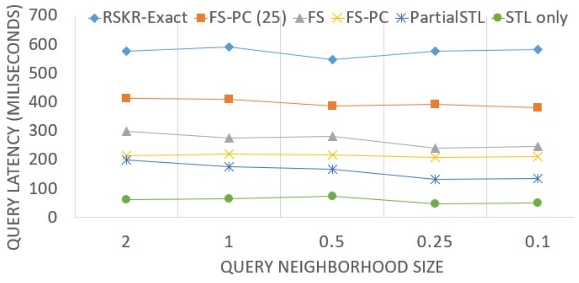
(RSK-W) is too slow (took almost an hour) hence is omitted from the figures. In both figures, we see significant performance improvement (note the logarithmic scale on the latency axis) with the introduction of the *horizontal jump* which drastically reduces the number of checked l-square neighborhoods.

We also observed in Figure 3.12a, that the query latency spikes for keywords whose rank is closer to k (in these experiments Q_{10}). When q is close to the k^{th} keyword (in rank), the jump size becomes smaller; as a result, more l-square neighborhoods are checked increasing the query latency for Q_{10}. For keywords ranked below Q_{10}, the jump size ($|f_q - f_k|$) is large because f_q is higher which leads to larger jumps and hence fewer l-square neighborhood checks improving the query latency. A similar justification explains the reduction of query latency for keywords with larger rank than Q_{10}; here f_q is much smaller but the jump size increases since it is an absolute value of the difference. Based on the these results, for the remaining experiments we will use the algorithm RSK-VH (Algorithm 8) for the (spatial) RSK query .

The next experiments consider the **RSKR query** performance using the same dataset (random sample with 10K tweets). Unless otherwise specified, both number of random restarts and number of divisions for each algorithm is set to 5. As in each coordinate division, we check



(a) Varying query keyword



(b) Varying neighborhood sizes

Figure 3.13: RSKR query latency.

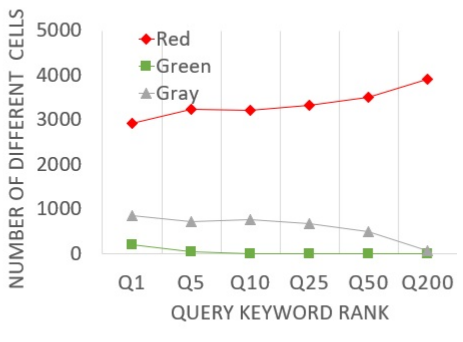
4 l-square neighborhoods, the budget per candidate cell was 100. We compare five approximate algorithms (i-v) and one exact algorithm (vi). These algorithms are: (i) Full STL (**FS**), which uses the STLs of the cells that are fully contained in the l-square neighborhood; (ii) Full STL and Previous Calculation (**FS-PC**), which uses the *termFreqMap* from previously checked l-square neighborhoods; (iii) **PartialSTL**; (iv) **STLOnly**, (v) **FS-PC(25)**, where the number of random restarts increased to 25; (vi) **RSK-Exact**, which is a variation of the RSK-VH algorithm that terminates processing a candidate cell as soon as one result is found for that cell. Figures 3.13b and 3.13a depict the query latency performance of the RSKR algorithms using different neighborhood sizes l (for keyword Q_5) and for different query keyword positions q (using $l = 0.5$) respectively. In both figures the RSK-Exact algorithm is the slowest among the RSKR algorithms since it explores the whole expansive region of a candidate cell, while all approximate algorithms use coordinate division to quickly focus to the part that is most likely to contain a result. Moreover, as expected, the latency for the RSKR query is much smaller (in msec) than the latency of the RSK query (in seconds).

In comparing the approximate algorithms, one has to also consider their precision and recall (to be examined later). In both figures we observe that the approximate algorithms' performance (higher to lower) is as follows : FS-PC(25) $>$ FS $>$ FS-PC $>$ PartialSTL $>$ STLOnly. Note that unless a result is found in a candidate cell, algorithm FS-PC(25) can randomly restart 25 times, while the other approximate algorithms can restart 5 times. Hence, when FS-PC(25) cannot find a result, it restarts more times making it the slowest (and also most accurate) approximate algorithm. Note that the difference between the FS and FS-PC algorithms is that the latter uses the $termFreqMap$ of the previously checked l-square neighborhood. Since $l \geq 2c$, there is a lot of overlap between two consecutive l-square neighborhoods. As a result, FS-PC is faster than FS since the use of the previous $termFreqMap$ allows it to take advantage of this overlap. PartialSTL and STLOnly do not check any tweet; instead they find results based on the STLs only, which adds another level of approximation. These algorithms are thus faster than the other approximate algorithms but their precision suffers from the extra approximation. STLOnly is the fastest (and has the lowest precision/recall as we will see in Figure 3.15) as it only considers the cells that are fully contained in the l-square neighborhood, completely ignoring the partially contained cells.

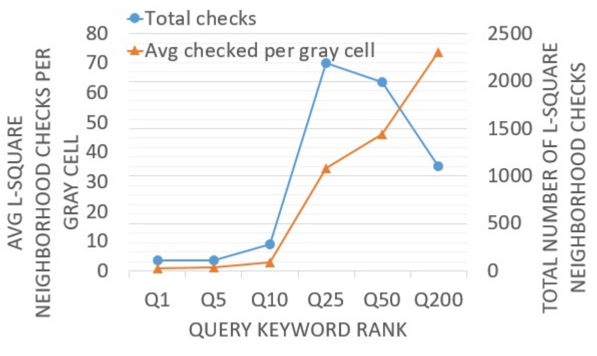
Another observation from Figure 3.13a is that RSK-Exact shows the same spike in query latency for the query positioned at rank k (Q_{10}), which is expected since it follows the RSK algorithm. We also observe that the behavior of the approximate algorithms is different, as it starts with a query latency decline from Q_1 to Q_5, followed by a maximum at Q_{25} and another decline until Q_{200}. There are two factors affecting this behavior, namely: (i) the number of candidate cells, and, (ii) the number of l-square neighborhoods checked. Figure 3.14a depicts the number of candidate (gray) cells as the rank of the query keyword increases. It also shows the number of red and green cells for reference purposes. Clearly, as the keyword rank increases the number of gray cells decreases. The sum of red, green and gray cells is constant for all keywords. As the keyword rank increases, the keyword becomes less popular and hence there are more cells for which we can easily decide that they have no answer, i.e., they become red cells. Since there are very few green cells, the numbers of gray and red cells change in opposite directions.

Figure 3.14b shows the (average) number of l-square neightborhoods checked per gray cell by algorithm FS-PC; the other approximate algorithms behave similarly. As the rank of the keyword increases, there are fewer possible answers (keyword is less popular) and thus we have to check more l-square neightborhoods to find an answer. The difference becomes more apparent in larger ranks since there are much fewer answers. Figure 3.14b also shows the total number of l-square neighborhoods that the FS-PC algorithm checks (over all gray cells); this is calculated by multiplying the number of gray cells in Figure 3.14a with the l-square neighborhoods checked per cell. This graph behaves similarly with the behavior seen in Figure 3.13a. From Q_1 to Q_5 the latency decreases since results are still easy to find but the number of candidate cells decreases and dictates the query latency. For Q_{10} the number of candidate cells decrease slightly in comparison with Q_5, however it is relatively harder to find an answer because the rank of the keyword is no longer less than k (and thus we have to check more l-square neighborhoods). This behavior continues until Q_{25}, after which the decline on the number of candidate cells dominates and reduces the query latency. Moreover, the higher ranked keywords have higher latency over the lower ranked keywords. As higher ranked keywords have much fewer answers, the approximate algorithm keeps checking random l-square neighborhoods (until it finds an answer or runs out of budget).

We compare the precision and recall of the different approximate algorithms for the RSKR query in Figures 3.15a and 3.15b respectively. While faster, the approaches that find results by looking only within STLs (namely the PartialSTL and STLOnly) suffer both in precision and recall

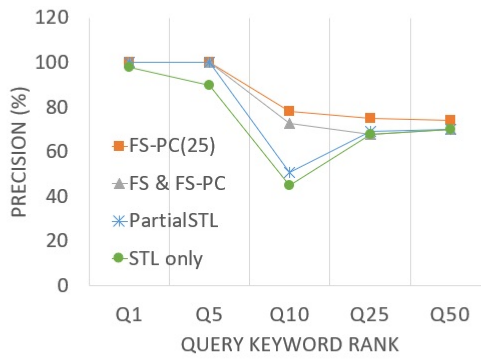


(a) Different cell counts.

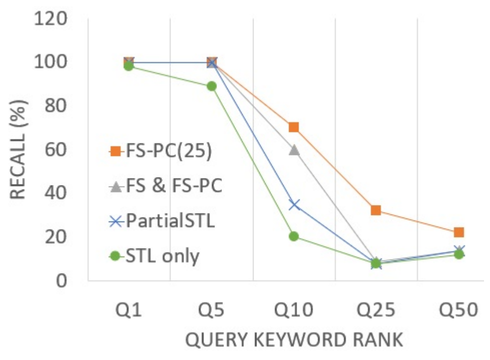


(b) Checked l-square neighborhoods.

Figure 3.14: Number of gray cells and checked l-square neighborhoods for different query keywords.



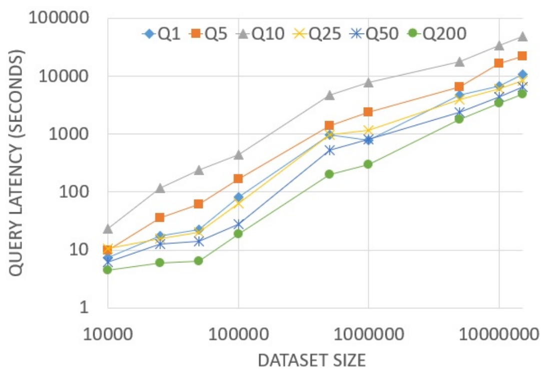
(a) Precision.



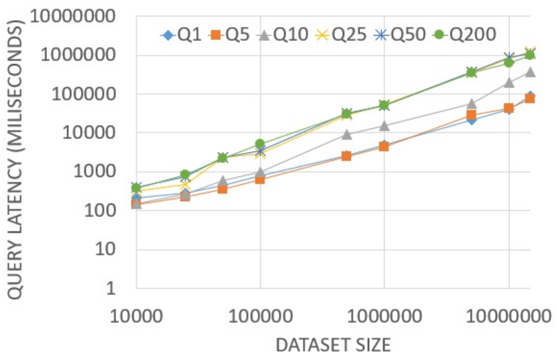
(b) Recall.

Figure 3.15: Precision and Recall of RSKR approximate algorithms for different keywords.

when compared to the other approximate algorithms. The precision of all the approximate algorithm is worst for Q_{10}, because its rank is equal to k. For the approximate algorithms, candidate cells for this particular keyword are the hardest to classify because the difference between the frequency of the k-th keyword and the frequency of Q_{10} is minimal. The FS and FS-PC algorithms have the same precision and recall since they use the same budget and differ only in the use of Previous Calculation. Overall, the algorithm using the Full STL and Previous Calculation (FS-PC) outperforms the other versions with respect to both precision and query latency. Hence, in the remaining experiments we will use FS-PC to answer RSKR query (Algorithm 9).



(a) RSK



(b) RSKR

Figure 3.16: Query latency while varying the dataset size.

Dataset Scalability: Having identified the best algorithm for each of the RSK and RSKR problems, we proceed with examining the effect of the dataset size. Figures 3.16(a) and (b) show the query latency for each problem respectively, for different keywords while varying the dataset sizes from 10K tweets to the full dataset of 15M tweets. We used neighborhood size, $l = 0.5$ and cell size, $c = 0.25$ for these experiments. We see similar trends for different query keywords for different sizes of dataset. Nevertheless, the query latency increases significantly with the increase of dataset size. This is expected as many more tweets must be processed. Even for the faster RSKR algorithm, the query latency becomes prohibitively large for big datasets (note the logarithmic scale). This leads us to explore scaling both the RSK and RSKR problems to multiple nodes.

3.7.4 Multi Node Evaluation

Effect of slicing. Our algorithms to answer the RSK and RSKR queries are highly parallelizable as each candidate (gray) cell can be processed independently. Moreover, the RSK algorithm slices each gray cell so that a single cell can be processed in parallel by multiple machines. In this part, we first examine the effectiveness of slicing; then we present scalability experiments for both problems. All

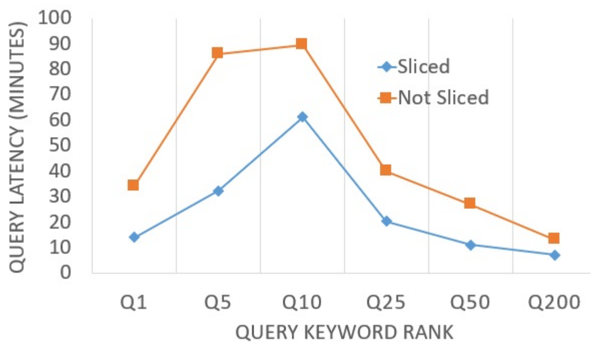
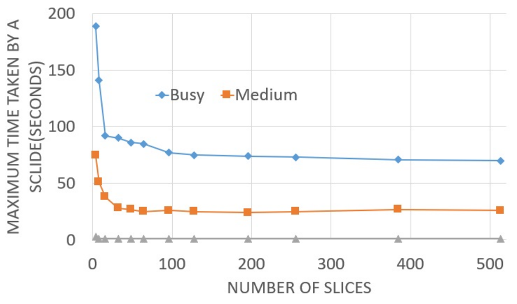


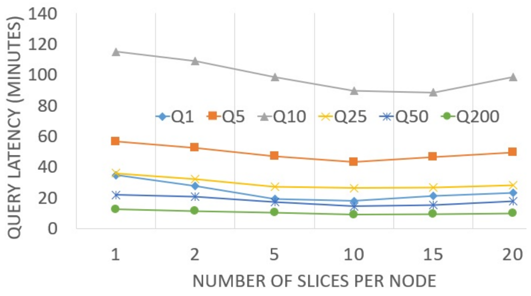
Figure 3.17: Comparing effect of slicing on query latency (in minutes) of RSK for different query keywords.

multi node experiments below were performed using the full 15M tweets dataset with neighborhood size l = 0.5 and cell size c = 0.25. To test the effectiveness of slicing, we experimentally evaluated two versions of the RSK algorithm without and with slicing (where we applied equal-post slicing using the approach discussed in Section 3.6). The results appear in Figure 3.17 for different query keyword positions q. We observe significant improvement in query latency by using slicing. This is because slicing provides better load balancing among the cores.

Since individual slices can be processed at different cores, the maximum time taken to process any of the slices is important. Figure 3.18a shows the relationship between the number of slices and the maximum processing time per slice. For this experiment we applied slicing to three different cells that we term *busy*, *medium* and *light*, based on their tweet density (with 303k, 3k and 587 tweets respectively). As the figure shows, slicing helps until a certain point, after which regardless of the number of slices, the maximum time taken by a slice does not improve. The reason there is a lower bound that we cannot go below is because we expand the region of each slice equal to the query neighborhood size l, so even the thinnest slice has to be expanded by l. This holds for all cell densities we experimented. As expected, slicing is more advantageous for busy (followed



(a) Max. time for a slice for Q_1.



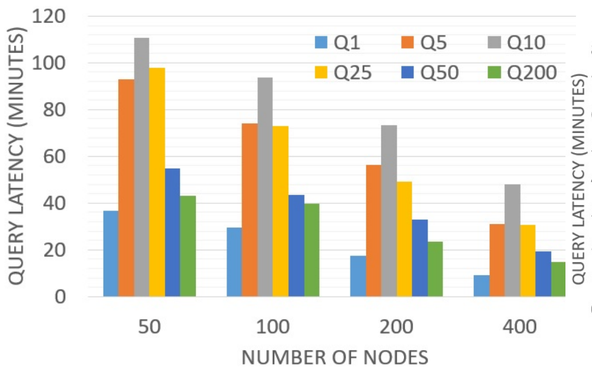
(b) Average slice count per core (γ)

Figure 3.18: Effect of slice count on query latency for different query keyword.

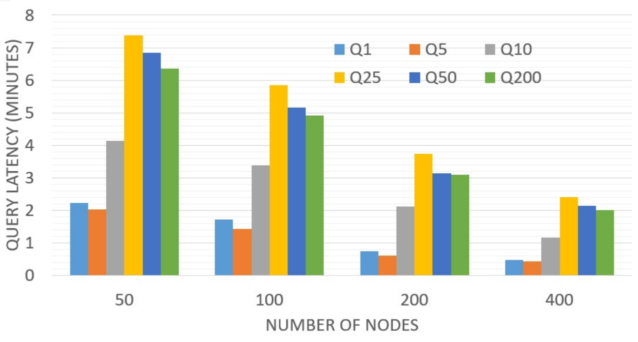
by medium and light) cells as it provides a larger reduction in latency. This asserts the findings of Figure 3.9 where we also showed that the number of posts is directly correlated with the time needed to process a cell.

We also examined the effect of γ, the average number of slices assigned to each core. Figure 3.18b shows the query latency while varying γ from 1 to 20, using a fixed number of cores ($\mathcal{M} = 300$). For all keywords, as we increase γ we increase the number of slices ($\gamma * \mathcal{M}$), which improves latency as it enables better distribution of workload. We observe again that more slicing (higher γ) is not going to help after a certain point when the time required per slice stops improving. Further adding slices per core will start deteriorating latency. This is because we always have to expand the slice by l regardless how thin the slice is. So slicing too much doesn't help. In the remaining experiments, we set γ = 10 as it depicted the best query latency.

Cluster Speed-up: To measure the speed-up performance we started with a cluster of 50 cores doubling them until 400 cores. Figure 3.19a shows the RSK query latency (using Algorithm 8) for different keywords while varying the number of cores in the cluster. For all queries, adding more cores improves the latency. The relative latencies of the keywords follow the same order as with the



(a) RSK



(b) RSKR

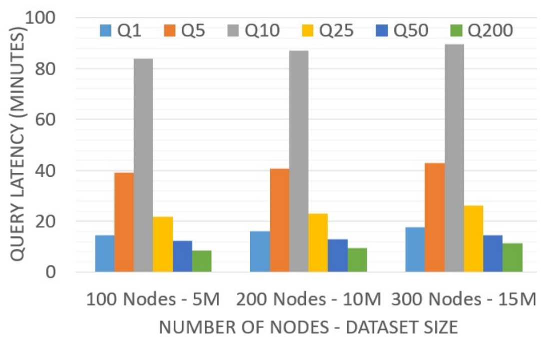
Figure 3.19: Query latency with varying number of cores for different keywords.

single-node experiments (Figure 3.12a). Figure 3.19b shows query latencies for RSKR (Algorithm 9) for different query keywords with different number of cores in the cluster. The query latencies for the less popular query keywords (Q_{25}, Q_{50}, Q_{200}) suffer for the same reason as mentioned for the single node experiments (Figure 3.13a).

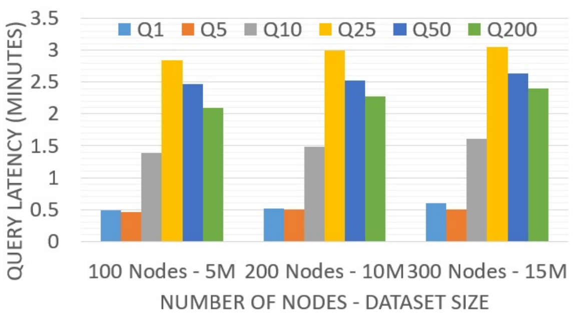
Cluster Scale-up: To explore the scale-up performance, we keep the workload constant (in terms of number of tweets per core) as we add more cores to the cluster. We start with a cluster of 100 cores and 5M tweets (i.e., 50K tweets/core) then 200 cores and 10M tweets and finally 300 cores and 15M tweets. Figure 3.20 shows query latency of RSK and RSKR (for different keywords) for the above scale-up experiments. Clearly both algorithms achieve very good scale-up performance; the query latency per keyword remains similar, which means that the additional data is processed in roughly the same amount of time if the cores are increased proportionately.

3.7.5 Comparison with GARNET

GARNET[42] has two fundamental limitations compared to our approach. Firstly, the neighborhood size in GARNET has to be equal to the cell size. Secondly, all the results have to be



(a) RSK



(b) RSKR

Figure 3.20: Scale-up experiments for different keywords.

aligned with the cell of the grid. Hence, GARNET provides an approximate answer to the RSKR (restricted) query. In particular, a cell is answer of the RSKR problem if there is at least one point in the cell that is (k, l)-*frequent*, whereas GARNET returns that cell if the center of the cell is (k, l)-*frequent*. Figure 3.21 depicts the total number of results returned by the two systems. For RSKR we depict both approaches, namely RSK-Exact and FS-PC. As expected, GARNET misses lots of results as it tests only the center of the cell while our approaches test numereous shifted l-square neighborhoods within each cell.

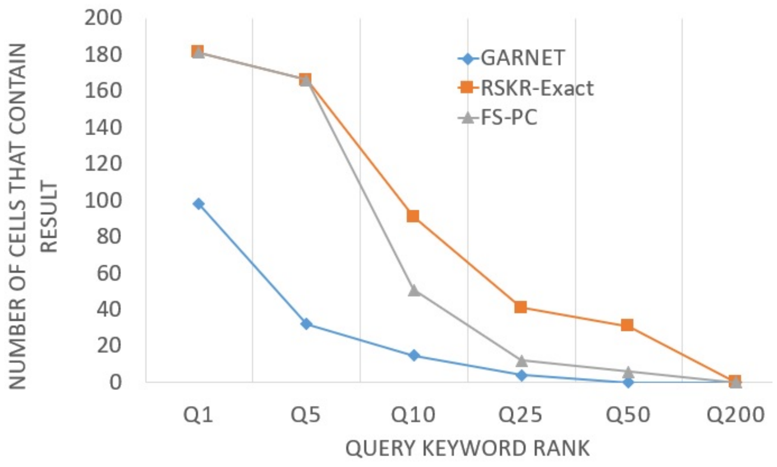


Figure 3.21: Comparison of number of results found by GARNET, RSK-Exact and FS-PC algorithm.

3.8 Conclusions and Future work

We introduced the Reverse Spatial Keyword (RSK) Query on geo-tagged posts that allows a user to identify where a particular keyword is popular. Using materialized term frequency lists we presented algorithms to solve RSK queries. We further proposed a restricted version of the query (RSKR) for which we present an exact and multiple faster but approximate algorithms. Parallelism is explored for the both exact and restricted problems. An interesting future direction is to explore RSK related queries over 3-dimensional spatiotemporal data.

Chapter 4

An Application: Reverse Spatial Top-k Snapshot Query over Social Media Streams

4.1 Introduction

In this chapter, we present an application of the RSK query over streaming social media data; this is a different environment than the archival setting used to present RSK in the previous chapter. Since typically an archived dataset is too large the RSK query also takes too long to process. However, there are many applications where the users are only interested in the most recent data, i.e., what people are talking about in the social media “right now”. The system presented in this chapter targets these types of applications. We formalize the problem as *Reverse Spatial Top-k Snapshot Query (RSKSQ)*, which like the RSK query identifies areas where a keyword is popular, however, it operates over the Twitter data stream. Given a query term q, an integer k, a neighborhood size l, a time window W, and a refresh rate r, we find the neighborhoods of size l where q is in the top-k most frequent terms among the tweets in those neighborhoods. To answer RSKSQ queries, we run the RSK query over a snapshot of a fixed-sized time window (W) of the most recent tweets, i.e., tweets posted in the last W minutes, and refresh the results every r time units. To implement this approach we use a circular queue $\mathcal{Q}$ of a fixed size of $\frac{W}{r}$ to keep track of the latest tweets. If $\mathcal{Q}$ becomes full, we discard the oldest tweets. To answer the RSKSQ, we use an index structure consisting of a uniform grid augmented by materialized lists of term frequencies and a filter-refinement-based RSK query processing algorithm optimized for fast updates. We have implemented a system that provides the results of RSKSQ using a desktop application based on ArcGIS[61]. Our contribution is as follows:

- We consider the *Reverse Spatial Top-k Snapshot Query (RSKSQ)*, which operates over the Twitter data stream. We implement it by running the Reverse Spatial Keyword (RSK) query on geo-tagged posts that arrived in time window of size W, and refresh the result every r time units.
- We present a working system using the live Twitter stream providing users with options to choose the size of the window W as well as the refresh rate i.e., the rate at which the result will be updated. The system is built as a desktop application based on ArcGIS [61].

The rest of the chapter is organized as follows: Section 4.2 discusses related work, while Section 4.3 formulates the RSKSQ system. Section 4.4 presents our proposed approach for the RSKSQ system. Various aspects and properties of our developed system is described in Section 4.5.

We explore the query latency of the our system, in Section 4.6, while conclusions and future work appear in Section 4.7.

4.2 Related Work

The most related works relate to **Continuous Monitoring over Data Streams**. In particular, [55] studies continuous monitoring of top-k queries over a fixed-size window over the most recent data. The window size can be expressed either in terms of the number of active tuples or time units. [25] investigates dynamic spatial–keyword objects whose locations and keywords change over time. They study the problem of continuously tracking top-k dynamic spatial–keyword objects for a given set of queries. A continuous query is issued once over a streaming dataset, and then logically runs continuously until it is terminated. It lets users get new incremental results from the dataset (when the dataset changes and the result is affected) without having to issue the same query repeatedly [13]. Our work is different from these because we focus on a snapshot query computed over posts contained over a time window rather than a continuous query. Moreover, these works basically return top-k objects as results while we return the spatial domain where the user provided keyword is among the top-k most frequent keywords.

4.3 Problem Definition

The **Reverse Spatial Top-k Snapshot Query (RSKSQ)** is defined by the tuple $\langle k, q, l, W, r \rangle$. That is, the query parameters required to answer the RSKSQ include all the parameters of an RSK query along with two additional parameters: (1) W the time-window that includes the posts we are interested in, and, (2) r the refresh rate. The refresh rate r is less than the time-window length

W; $r < W$. Typical values of r are in seconds while normally W is in minutes. RSKSQ queries identify areas where a keyword is popular among the posts included within the time-window, W and the results are refreshed every r time units. That is, the answer to RSKSQ is the set of spatial regions where q is (k, l)-*frequent* at each point in these regions within the time-window W.

4.4 Our Approach

Our approach to build a system for the RSKSQ query consists of two parts: (1) **Storing and Updating the dataset**. This implies efficiently storing the tweets originating during the current time window W and then updating this set of tweets after every r period of time; this step is described in more details in Section 4.4.2. (2) **Answer the RSK Query**. We use a filering and refinement based RSK algorithm to answer the Q_{RSK} part in Q_{RSKSQ} every r time units.

4.4.1 System Architecture

The overall system is illustrated in Figure 4.1. First, the new batch of tweets arrive to the Ring Buffer ($\mathcal{B}$) which is updated to keep the tweets within the W time window. Then, the STL augmented Grid is updated from $\mathcal{B}$ where each cell's tweets and its STL are updated. After that, the RSK Query Processor starts working where the Grid acts as its input. Finally, the result is projected on the map for the users. This whole process is repeated every r time units. Users have the opportunity to update the query keyword q and k.

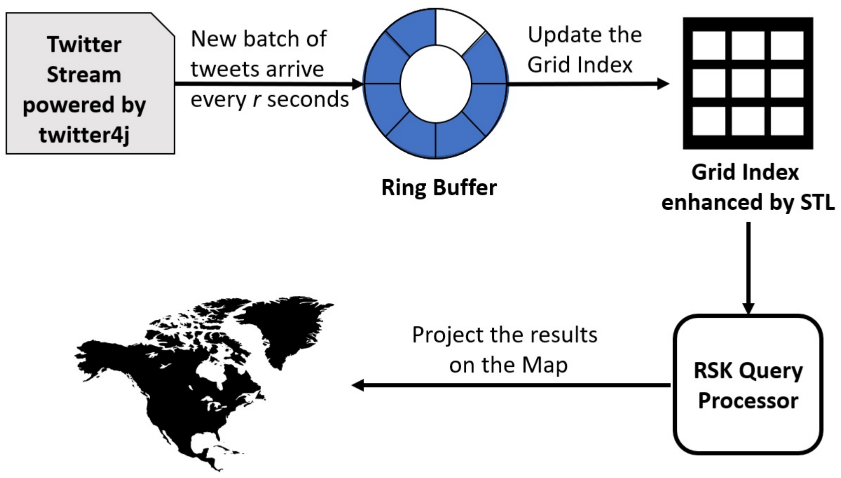

Twitter Stream powered by twitter4j
New batch of tweets arrive every r seconds
Ring Buffer
Update the Grid Index
Grid Index enhanced by STL
RSK Query Processor
Project the results on the Map

Figure 4.1: RSKSQ System Architecture.

4.4.2 Data Storage

The RSKSQ system has to update its results every r time units (typically in seconds). It is vital to design a storage system for the tweets that is fast, efficient and easy to update. A good design should have a low overall network bandwidth requirement. While updating the storage, each tweet should be downloaded only once. We use a Circular Queue or Ring Buffer of size $\frac{W}{r}$; each element in the buffer stores the set of tweets fetched every r time units. Once the buffer is full, i.e., we have tweets for the last W time interval, we discard the oldest set of tweets from the buffer and add the latest set of tweets in its place. Figure 4.2 shows how new data arriving every r time units is stored in the ring buffer. The ring buffer's elements are empty at first (shown in blue). As more and more batches of tweets arrive (Figure 4.2(a)-(h)), the cells fill up (shown in white). Once the whole buffer is filled, we insert the next batch of tweets into the element which holds the oldest set of tweets as shown in Figure 4.2(i). As a result, each tweet post is downloaded only once. Moreover, this technique keeps the in-memory database as small as possible by only holding the tweets from the last W time interval. We get the whole dataset ($\mathcal{D}$) by merging the non-empty sets of tweets from the ring buffer $\mathcal{B}$.

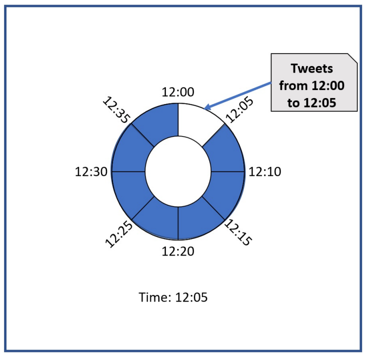
4.4.3 Efficient Updating the Indices and STLs

We optimize the STL calculation by only updating the STLs of the cells whose tweet set has been changed, after every refresh of the dataset (every r time units). If a cell has any change in its set of tweets, we keep track of the set of tweets added from the newest batch of tweets. We also keep track of the set of tweets removed from a cell from the oldest set of tweets removed (in case of full Ring Buffer $\mathcal{B}$). We use these two lists of tweets to update the STL of the cell without reprocessing the terms of all the tweets in the cell every time the results are updated.

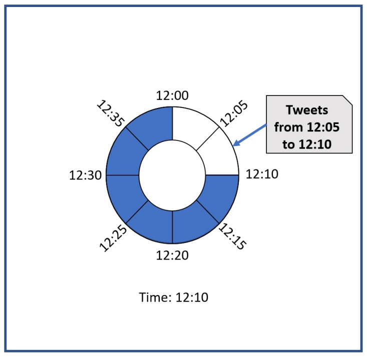
4.5 System Description

Hardware: We host the demo system in single-node environment, run on a machine featuring an Intel Core i-7 processor (8 cores) with 16 GB of RAM and 512 GB solid state drive. The RSK Query uses all available (eight) cores on the processor.

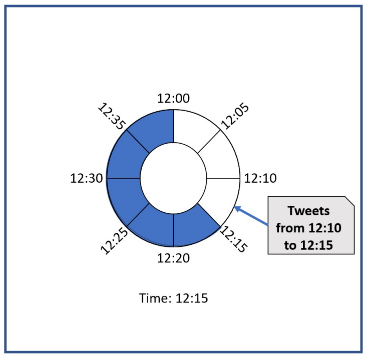
Data Storage and Dataset: We implement the Ring Buffer ($\mathcal{B}$) as a list of sets of tweets while we sort these tweets by their arrival time in the buffer. We use the Twitter streaming API of twitter4j [68] to collect all geo-tagged (i.e., tweets that have the user's GPS location or the user's 'Twitter Place'), English-based tweets (i.e., excluding empty tweets or tweets with only URLs) from a rectangle that contains the New York state and surrounding areas (i.e., region A has GPS coordinates: -76, -69, 43, 36). As before, since there are only about 5% tweets with actual GPS location, for the rest of the geo-tagged tweets (i.e., those with the user's 'Twitter Place') we used as location, a random point



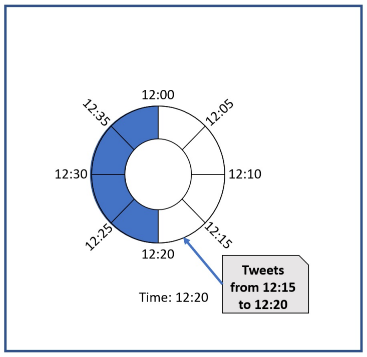
(a) Ring Buffer at 12:05 PM.



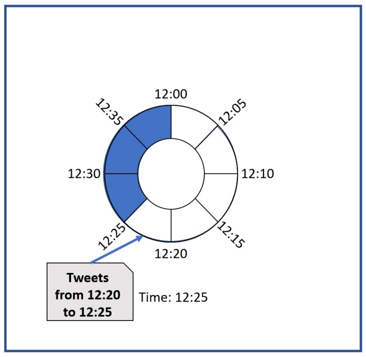
(b) Ring Buffer at 12:10 PM.



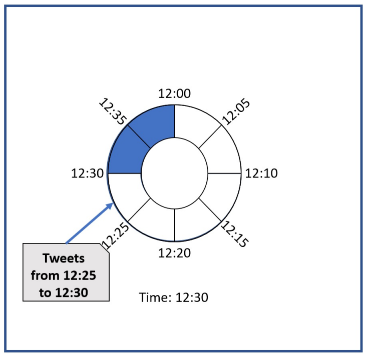
(c) Ring Buffer at 12:15 PM.



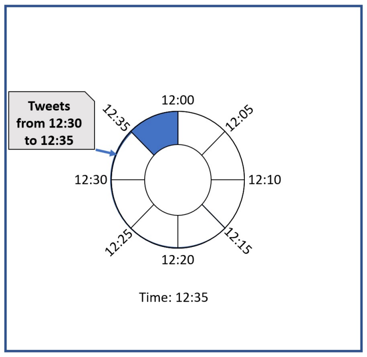
(d) Ring Buffer at 12:20 PM.



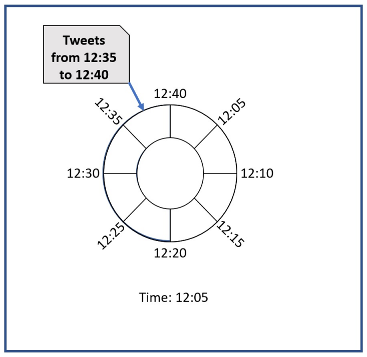
(e) Ring Buffer at 12:25 PM.



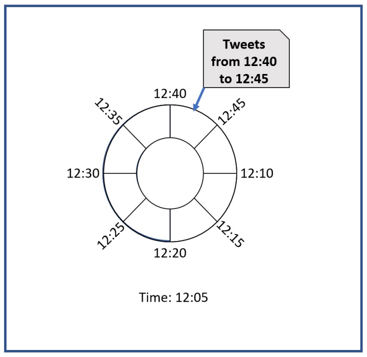
(f) Ring Buffer at 12:30 PM.



(g) Ring Buffer at 12:35 PM.



(h) Ring Buffer at 12:40 PM.

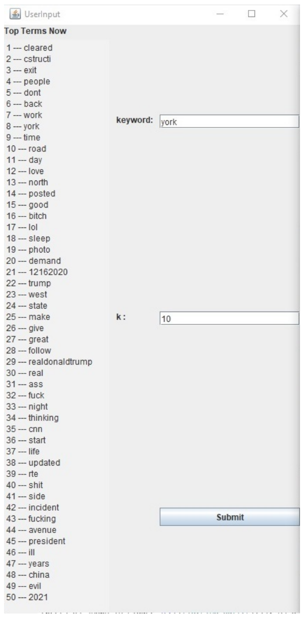


(i) Ring Buffer at 12:45 PM.

Figure 4.2: Ring Buffer at different time where refresh rate r = 300 seconds and Window size W = 40 minutes.

in the provided polygon that the Twitter API shares for ‘Place’. Each tweet record has the tweet’s spatio-temporal coordinates and its terms. We also remove the stopwords to get more useful results.

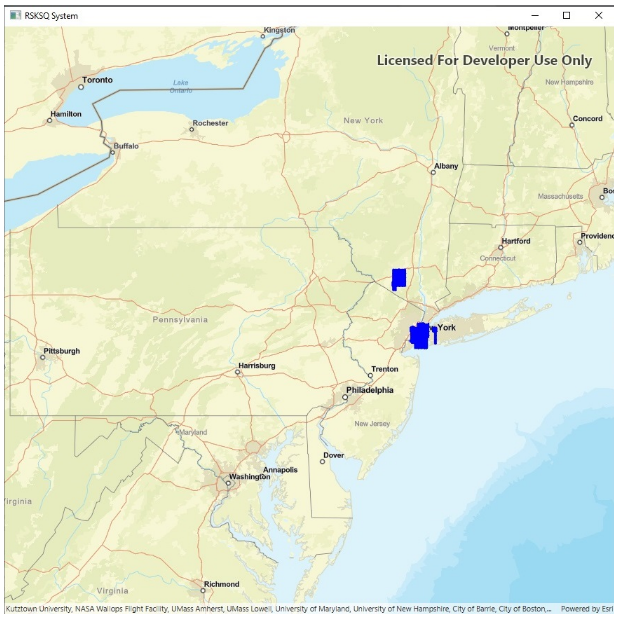
Index Structure: We divided the space covered by each dataset into a uniform grid of square cells with each cell containing tweets that are in the geographical area covered by that cell. Each cell is enhanced with a STL of its tweets. We update the index structure every r time units, by removing the tweets from the oldest set and adding newly arriving tweets. If there is change of tweets in a cell, we update that cell’s STL as well.



```
UserInput
Top Terms Now
1 --- cleared
2 --- cstructi
3 --- exit
4 --- people
5 --- dont
6 --- back
7 --- work
8 --- york
9 --- time
10 --- road
11 --- day
12 --- love
13 --- north
14 --- posted
15 --- good
16 --- bitch
17 --- lol
18 --- sleep
19 --- photo
20 --- demand
21 --- 12162020
22 --- trump
23 --- west
24 --- state
25 --- make
26 --- give
27 --- great
28 --- follow
29 --- realdonaldtrump
30 --- real
31 --- ass
32 --- fuck
33 --- night
34 --- thinking
35 --- cnn
36 --- start
37 --- life
38 --- updated
39 --- rte
40 --- shit
41 --- side
42 --- incident
43 --- fucking
44 --- avenue
45 --- president
46 --- ill
47 --- years
48 --- china
49 --- evil
50 --- 2021

keyword: york
k : 10
```

(a) User Input interface.


RSKSQ System
Licensed For Developer Use Only
Kingston
Toronto
Hamilton
Rochester
Buffalo
New York
Albany
Concord
Hartford
Providence
Pennsylvania
Pittsburgh
Harrisburg
York
Trenton
Philadelphia
New Jersey
Dover
Annapolis
Washington
Richmond
Virginia
Kutztown University, NASA Wallops Flight Facility, UMass Amherst, UMass Lowell, University of Maryland, University of New Hampshire, City of Barrie, City of Boston,... Powered by Esri

(b) Output over map.

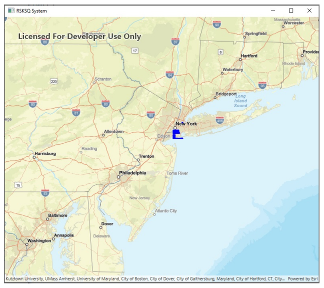
Figure 4.3: User Interface of the RSKSQ system.

User Interface: The user interface of our RSKSQ system is shown in Figure 4.3. It has two parts, (1) User Input interface shown in Figure 4.3a, and (2) Output over Map shown in Figure 4.3b. In the user input interface, there is a list of 50 most frequent terms in the current time window along with their respective ranks. This list is updated every r time units just like the output in the map. The user input interface allows the user to update the query keyword (q) and the value of k. The output will change according to the new query keyword (q) and k. The top term list on the left side of the input interface can act as a guide to find useful results using our RSKSQ system. As the ranks of the terms in the list are from all the tweets in the current snapshot, we can find the locations where a particular keyword (q) is more popular by changing the value of k to be equal to the rank of that keyword. The output of the RSKSQ system will show the places where the given query keyword is more popular than other places in the current time window.

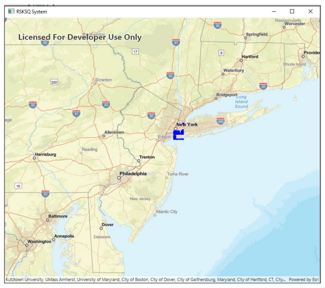
Output on Map: Figure 4.4, 4.5 and 4.6 show the output of our RSKSQ system over real data from the live Twitter stream. For all the experiments here, we have used a time window W = 2 hours. On Sunday, December 6th, 2020, we ran the query q = "$brooklyn$" with k = 10 (in the same geographic area described before). As expected, the keyword is popular around Brooklyn in the New York state. We run the same query again on December 17th and find very similar result. As the keyword "brooklyn" is a name of a place, it's popularity/frequency remains the similar unless there is some trending event related to that place.

On the same day, December 6th, we ran the query q = "$rudigiuliani$" with k = 50, which coincides with the news that then US President's attorney Rudi Giuliani had been diagnosed with covid19. We can see from the output shown in Figure 4.5 (a)-(c), that the keyword is popular around various locations in the New York state. We run the same query again on the same time but on a different day, on December 17th. This time around we see from the output shown in Figure 4.5(d)-(f) that there are actually way fewer results (if any). This is because the news is not trending anymore and the query keyword itself is not popular enough to be among the top-50 most frequent terms in many areas.

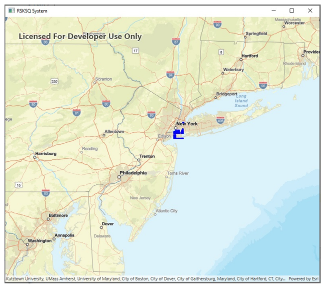
We further run the query q = "$pfizer$" on December 15th, 2020 with k = 100. Pfizer is the name of the company whose vaccine has been recently approved for use in USA. The results are shown in Figure 4.6. As is apparent from the result, the keyword's popularity is unchanged over time as it a frequently discussed topic.



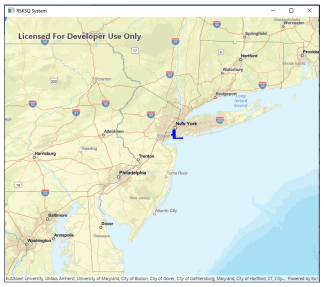
(a) December 6th, 9:39 PM.



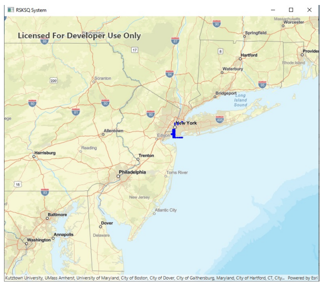
(b) December 6th, 9:43 PM.



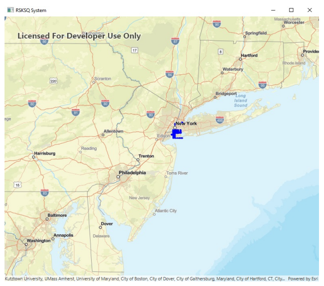
(c) December 6th, 9:48 PM.



(d) December 17th, 9:39 PM.



(e) December 17th, 9:43 PM.

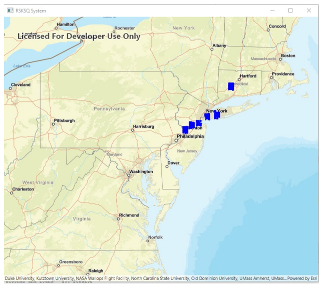


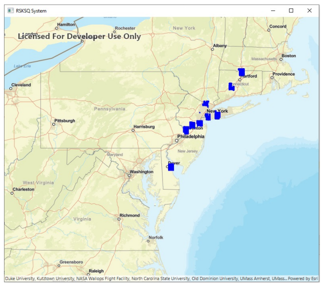
(f) December 17th, 9:48 PM.

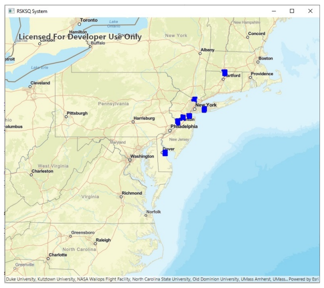
Figure 4.4: Output of RSKSQ system for q = "$brooklyn$" and k = 10 from various times on December 6th and December 17th, 2020.

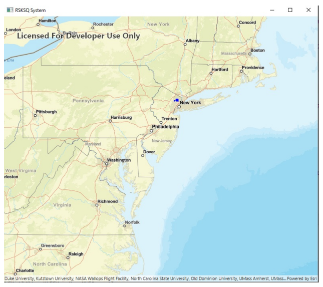
4.6 Experiments

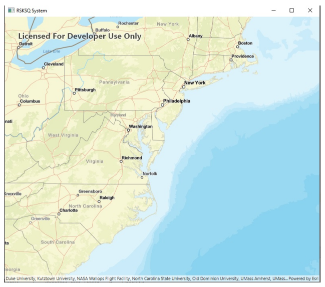
In this section, we explore the query latency of the RSKSQ system, which clearly depends on the size of the dataset within the W window. The size of the dataset depends on: (1) time of day (during daytime for the same W, we get many more tweets), and (2) size of time window W. We run the experiment during daytime when the rate of ingestion of tweets is highest. In particular,


(a) December 6th, 8:00 PM.


(b) December 6th, 8:08 PM.


(c) December 6th, 8:12 PM.


(d) December 17th, 8:00 PM.


(e) December 17th, 8:08 PM.

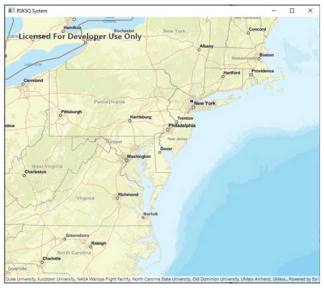
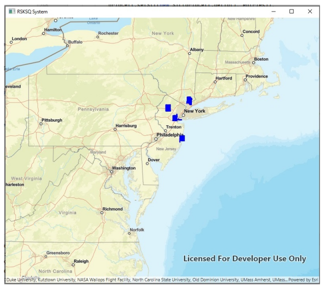
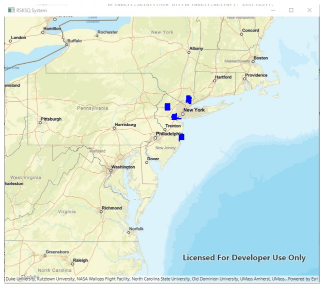
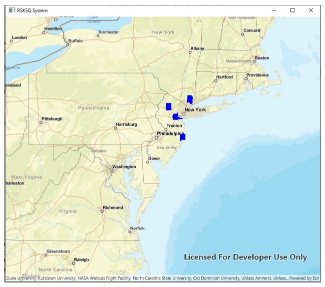
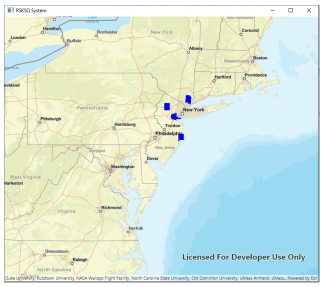

(f) December 17th, 8:12 PM.

Figure 4.5: Output of RSKSQ system for q = "$rudigiuliani$" and k = 50 from various times in the evening of December 6th and December 17th, 2020.


(a) Output at 8:30 PM.


(b) Output at 8:32 PM.


(c) Output at 8:34 PM.


(d) Output at 8:36 PM.

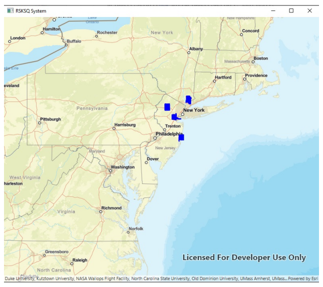
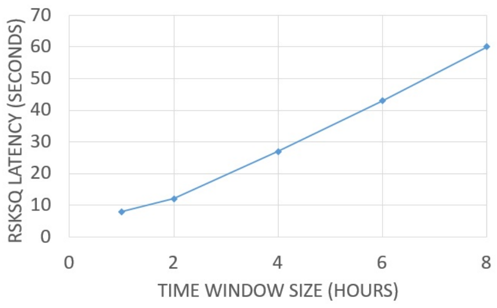
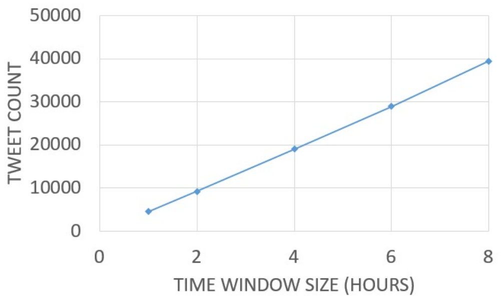

(e) Output at 8:40 PM.

Figure 4.6: Output of RSKSQ system for q = "$pfizer$" and k = 100 from various times in the evening of December 15th, 2020.

the experiment was conducted using the live Twitter stream starting from December 16^{th}, 2020 at 7 AM PST. For this experiment, we use the query keyword q whose rank is equal to k; this is because in this case the query latency is highest for the RSK query as discussed and experimentally shown in Chapter 3. The query neighborhood side length (l) is 0.2 and the cell size (c) is 0.05. Figure 4.7 shows how query latency and the number of tweets change with the change of window size W. As expected, both query latency and tweet count increase with the increase of the window size W. Note that the query latency indicates the smallest possible value of r for the respective window size W.



(a) RSKSQ latency.



(b) Tweet count.

Figure 4.7: Query latency and tweet count for varying the size of time window W.

4.7 Conclusion and Future Work

We introduce the Reverse Spatial Top-k Snapshot Query (RSKSQ) system on geo-tagged posts that allows a user to identify where a particular keyword is popular over the live Twitter stream. Using materialized term frequency lists and a ring buffer based main memory storage, we develop a system to answer the RSKSQ query. An interesting future direction is to explore ways to extend our system to support continuous queries over the live Twitter stream.

Chapter 5

Conclusions

The wide availability of tracking devices has drastically increased the role of geolocation in social networks, resulting in new applications. This has led to new challenges as well as opportunities to explore and get better insight about people on the social media. There has been numerous works on how to use these information in the best way possible. In these thesis, we explore several avenues of this promising field of study.

In chapter 2, 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 (kFST). 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. In chapter 3, we introduce the Reverse Spatial Keyword (RSK) query on geo-tagged posts that allows a user to identify where a particular keyword is popular. Using materialized term frequency lists we present algorithms to solve RSK queries. We further propose a restricted version of the query (RSKR) for which we present an exact and multiple faster but approximate algorithms. Parallelism is explored for both exact and restricted problems. In chapter 4, we introduce the Reverse Spatial Top-k Snapshot Query (RSKSQ) on geo-tagged posts that allows a user to identify where a particular keyword is popular over live Twitter stream. Using materialized term frequency lists and ring buffer based main memory storage, we develop a system to answer RSKSQ query.

As for future work, there are several interesting directions. One would be to enhance the STL approach (presented in chapter 2) with a distributed threshold algorithm (like [16]) so as to process even larger volumes of data. Another direction would be to explore RSK related queries over spatio-temporal data (i.e. 3-dimensional) instead of spatial data which we discussed in chapter 3. It would also be very interesting to extend our system presented in chapter 4 to support continuous queries over the live Twitter stream along with snapshot queries which is already supported in our work.

Bibliography

[1] Geofeedia, https://geofeedia.com/.

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

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

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

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

[6] *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2009.

[7] *Proceedings of the 26th International Conference on Data Engineering,ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. IEEE Computer Society, 2010.

[8] *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia - July 06 - 11, 2014*. ACM, 2014.

[9] Elke Achtert, Christian Böhm, Peer Kröger, Peter Kunath, Alexey Pryakhin, and Matthias Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 515—-526. ACM, 2006.

[10] Pritom Ahmed, Mahbub Hasan, Abhijith Kashyap, Vagelis Hristidis, and Vassilis J. Tsotras. Efficient computation of top-k frequent terms over spatio-temporal ranges. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference*. ACM, 2017.

[11] Ritesh Ahuja, Nikos Armenatzoglou, Dimitris Papadias, and George J Fakas. Geo-social keyword search. In *International Symposium on Spatial and Temporal Databases*, pages 431–450. Springer, 2015.

[12] Nikos Armenatzoglou, Ritesh Ahuja, and Dimitris Papadias. Geo-social ranking: functions and query processing. *VLDB J.*, 24(6):783–799, 2015.

[13] Shivnath Babu. *Continuous Query*, pages 492–493. Springer US, Boston, MA, 2009.

[14] H. Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. IO-Top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[15] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD*, pages 322–331, 1990.

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

[17] Xin Cao, Gao Cong, Tao Guo, Christian S. Jensen, and Beng Chin Ooi. Efficient processing of spatial group keyword queries. *ACM Trans. Database Syst.*, 40(2):13, 2015.

[18] Xin Cao, Gao Cong, and Christian S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.

[19] Xin Cao, Gao Cong, Christian S. Jensen, Jun Jie Ng, Beng Chin Ooi, Nhan-Tue Phan, and Dingming Wu. SWORS: A system for the efficient retrieval of relevant spatial web objects. *PVLDB*, 5(12):1914–1917, 2012.

[20] Ho-Leung Chan, Tak Wah Lam, Lap-Kei Lee, and Hing-Fung Ting. Continuous monitoring of distributed data streams over a time-based sliding window. *Algorithmica*, 62(3-4):1088–1111, 2012.

[21] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[22] Zhiyuan Cheng, James Caverlee, and Kyumin Lee. You are where you tweet: A content-based approach to geo-locating twitter users. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management, CIKM*. ACM, 2010.

[23] Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors. *Proceedings of the 23rd International Conference on Data Engineering,ICDE 2007*. IEEE Computer Society, 2007.

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

[25] Yuyang Dong, Chuan Xiao, Hanxiong Chen, Jeffrey Xu Yu, Kunihiro Takeoka, Masafumi Oyamada, and Hiroyuki Kitagawa. Continuous top-k spatial–keyword search on dynamic objects. *The VLDB Journal*, pages 1–21, 2020.

[26] Leo Egghe. Untangling herdan's law and heaps' law: Mathematical and informetric arguments. *JASIST*, 2007.

[27] Facebook, https://www.facebook.com/.

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

[29] S. Farazi and D. Rafiei. Top-k frequent term queries on streaming data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2019.

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

[31] Foursquare, https://foursquare.com/.

[32] Yunjun Gao, Xu Qin, Baihua Zheng, and Gang Chen. Efficient reverse top-k boolean spatial keyword queries on road networks. *IEEE Trans. Knowl. Data Eng.*, 2015.

[33] Shen Ge, Leong Hou U, Nikos Mamoulis, and David W. Cheung. Efficient all top-(k) computation—a unified solution for all top-(k), reverse top-(k) and top-(m) influential queries. *IEEE Trans. on Knowl. and Data Eng.*, 2013.

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

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

[36] Marios Hadjieleftheriou, George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On-line discovery of dense areas in spatio-temporal databases. In *Advances in Spatial and Temporal Databases*, 2003.

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

[38] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 1961.

[39] Giacomo Inches, Mark J. Carman, and Fabio Crestani. Statistics of online user-generated short documents. In *ECIR*, pages 649–652, 2010.

[40] Instragam, https://www.instagram.com/.

[41] Mike Izbicki, Vagelis Papalexakis, and Vassilis J. Tsotras. Geolocating tweets in any language at any location. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM*. ACM, 2019.

[42] Christopher Jonathan, Amr Magdy, Mohamed F. Mokbel, and Albert Jonathan. GARNET: A holistic system approach for trending queries in microblogs. *32nd IEEE ICDE*, pages 1251–1262, 6 2016.

[43] Dimitrios Kotsakos, Theodoros Lappas, Dimitrios Kotzias, Dimitrios Gunopulos, Nattiya Kanhabua, and Kjetil Nørvåg. A burstiness-aware approach for document dating. In *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia- July 06 - 11, 2014*, 2014.

[44] Theodoros Lappas, Benjamin Arai, Manolis Platakis, Dimitrios Kotsakos, and Dimitrios Gunopulos. On burstiness-aware search for document sequences. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[45] Theodoros Lappas, Marcos R. Vieira, Dimitrios Gunopulos, and Vassilis J. Tsotras. On the spatiotemporal burstiness of terms. *Proc. VLDB Endow.*, 5(9):836–847, May 2012.

[46] Theodoros Lappas, Marcos R. Vieira, Dimitrios Gunopulos, and Vassilis J. Tsotras. On the spatiotemporal burstiness of terms. *PVLDB*, 2012.

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

[48] Y Liu, L Chen, N Jing, and L Liu. Parallel top-k spatial join query processing on massive spatial data. *J Comput Res Dev*, 2011.

[49] Jiaheng Lu, Ying Lu, and Gao Cong. Reverse spatial and textual k nearest neighbor search. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11. ACM, 2011.

[50] Changyin Luo, Junlin Li, Guohui Li, Wei Wei, Yanhong Li, and Jianjun Li. Efficient reverse spatial and textual k nearest neighbor queries on road networks. *Knowl.-Based Syst.*, 2016.

[51] C. Ma, H. Lu, L. Shou, and G. Chen. Ksq: Top-k similarity query on uncertain trajectories. *IEEE Transactions on Knowledge and Data Engineering*, 2013.

[52] Amr Magdy, Ahmed Aly, Mohamed Mokbel, Sameh Elnikety, Yuxiong He, Suman Nath, and Walid Aref. GeoTrend: Spatial trending queries on real-time microblogs. *ACM GIS Conf.*, 2016.

[53] N. Mamoulis, Kit Hung Cheng, Man Lung Yiu, and D. W. Cheung. Efficient aggregation of ranked inputs. In *IEEE ICDE*, pages 72–72, April 2006.

[54] Michael Mathioudakis, Nilesh Bansal, and Nick Koudas. Identifying, attributing and describing spatial bursts. *PVLDB*, 2010.

[55] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 635–646. ACM, 2006.

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

[57] Jinfeng Ni and Chinya V. Ravishankar. Pointwise-dense region queries in spatio-temporal databases. In *ICDE*. IEEE Computer Society, 2007.

[58] Jinfeng Ni and Chinya V. Ravishankar. Pointwise-dense region queries in spatio-temporal databases. In *Proceedings of the 23rd International Conference on Data Engineering,ICDE*, 2007.

[59] P. Nikitopoulos, G. A. Sfyris, A. Vlachou, C. Doulkeridis, and O. Telelis. Parallel and distributed processing of reverse top-k queries. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019.

[60] Baiyou Qiao, Bing Hu, Junhai Zhu, Gang Wu, Christophe Giraud-Carrier, and Guoren Wang. A top-k spatial join querying processing algorithm based on spark. *Information Systems*, 2020.

[61] CA: Environmental Systems Research Institute Redlands. Arcgis desktop: Release 10.8, 2020.

[62] João B. Rocha-Junior, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. Efficient processing of top-k spatial preference queries. *PVLDB*, 2010.

[63] João B. Rocha-Junior, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. Efficient processing of top-k spatial preference queries. *PVLDB*, 4(2):93–104, 2010.

[64] R. S. Saranya and S. Saraswathi. Ranking spatial data by quality preferences. In *IEEE-International Conference On Advances In Engineering, Science And Management (ICAESM -2012)*, 2012.

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

[66] Stefan Stieglitz, Milad Mirbabaie, Björn Ross, and Christoph Neuberger. Social media analytics–challenges in topic discovery, data collection, and data preparation. *International journal of information management*, 39, 2018.

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

[68] Twitter4j, http://twitter4j.org/.

[69] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. Reverse top-k queries. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. IEEE Computer Society, 2010.

[70] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. Monochromatic and bichromatic reverse top-k queries. *IEEE Trans. Knowl. Data Eng.*, 2011.

[71] Akrivi Vlachou, Christos Doulkeridis, Kjetil Nørvåg, and Yannis Kotidis. Identifying the most influential data objects with reverse top-k queries. *Proc. VLDB Endow.*, 2010.

[72] Akrivi Vlachou, Christos Doulkeridis, Kjetil Nørvåg, and Yannis Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13. ACM, 2013.

[73] Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. Monitoring reverse top-k queries over mobile devices. In *Proceedings of the 10th ACM International Workshop on Data Engineering for Wireless and Mobile Access*. Association for Computing Machinery, 2011.

[74] S. Wang, Z. Bao, J. S. Culpepper, T. Sellis, M. Sanderson, and X. Qin. Answering top-k exemplar trajectory queries. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2017.

[75] Xiaoyang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Wei Wang. Selectivity estimation on streaming spatio-textual data using local correlations. *Proc. VLDB Endow.*, 2014.

[76] Wolframalpha, https://www.wolframalpha.com/.

[77] Dingming Wu, Gao Cong, and Christian S. Jensen. A framework for efficient spatial web object retrieval. *VLDB J.*, 21(6):797–822, 2012.

[78] Dingming Wu, Man Lung Yiu, Gao Cong, and Christian S. Jensen. Joint top-k spatial keyword query processing. *IEEE Trans. Knowl. Data Eng.*, 24(10):1889–1903, 2012.

[79] Dingming Wu, Man Lung Yiu, Christian S. Jensen, and Gao Cong. Efficient continuously moving top-k spatial keyword query processing. In *IEEE ICDE*, pages 541–552, 2011.

[80] Tian Xia, Donghui Zhang, Evangelos Kanoulas, and Yang Du. On computing top-t most influential spatial sites. In *Proceedings of the 31st International Conference on Very Large Data Bases*, page 946–957. VLDB Endowment, 2005.

[81] M. L. Yiu, X. Dai, N. Mamoulis, and M. Vaitis. Top-k spatial preference queries. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1076–1085. IEEE Computer Society, 2007.

[82] Man Lung Yiu, Xiangyuan Dai, Nikos Mamoulis, and Michail Vaitis. Top-k spatial preference queries. In Chirkova et al. [23], pages 1076–1085.

[83] Manli Zhu, Dimitris Papadias, Dik Lun Lee, and Jun Zhang. Top-k spatial joins. *IEEE Transactions on Knowledge and Data Engineering*, 2005.