The VLDB Journal (2023) 32:501–524
https://doi.org/10.1007/s00778-022-00759-9

REGULAR PAPER

Reverse spatial top-k keyword queries

Pritom Ahmed[1] · **Ahmed Eldawy**[1] · **Vagelis Hristidis**[1] · **Vassilis J. Tsotras**[1]

Received: 21 June 2021 / Revised: 13 March 2022 / Accepted: 20 June 2022 / Published online: 25 July 2022

Abstract

We introduce the ***R**everse **S**patial Top-k **K**eyword (RSK)* query, which is defined as: *given a query term q, an integer k and a neighborhood size find all 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 a given cell size 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 with the grid. Furthermore, for every neighborhood (square) that is an answer, we can define infinitely more result neighborhoods by minimally shifting the square without including more posts in it. To address that, we need to identify contiguous regions where any point in the region can be the center of a neighborhood that satisfies the query. We propose an algorithm to efficiently answer an RSK query using an index structure consisting of a uniform grid augmented by materialized lists of term frequencies. We apply various optimizations that drastically improve query latency against baseline approaches. We also provide a theoretical model to choose the optimal cell size for the index to minimize query latency. We further examine a restricted version of the problem (RSKR) that limits the scope of the answer and propose efficient *approximate* algorithms. Finally, we examine how parallelism can improve performance by balancing the workload using a smart *load slicing* technique. Extensive experimental performance evaluation of the proposed methods using real Twitter datasets and crime report datasets, shows the efficiency of our optimizations and the accuracy of the proposed theoretical model.

Keywords Top-K · Reverse top-K · Spatial data · Social networks

1 Introduction

The wide availability of tracking devices has drastically increased the role of geolocation in social networks. Several online social sites, such as Twitter [39], Instagram [17] and Foursquare [13], are allowing users to geotag their social posts. This creates novel data analytic problems, such as detecting popular topic trends [20], most frequent trajectories [36], etc. These previous works apply various top-k algorithms in the spatio-temporal domain: given a query region, find the top-k most frequent terms, trajectories, etc. In this paper, 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, the query size and time window can be adjusted. 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). Using geo-located crime datasets as we show

✉ Pritom Ahmed
pahme002@ucr.edu

Ahmed Eldawy
eldawy@cs.ucr.edu

Vagelis Hristidis
vagelis@cs.ucr.edu

Vassilis J. Tsotras
tsotras@cs.ucr.edu

[1] University of California Riverside, Riverside, CA, USA

later in the paper, we can find areas where a particular crime is more common/frequent. 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 paper, 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). A neighborhood is unique for our problem if the posts inside the neighborhood's region are different. As a result, for every post, it is theoretically possible to have $(N-1)$ unique neighborhoods. As a result, any pair of coordinates from two points can be a corner for a unique neighborhood for a total of N^2 pairs. 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 filtering 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 paper is organized as follows: Sect. 2 discusses related work, while Sect. 3 formulates the RSK and RSKR queries. Section 4 presents our algorithms for the RSK and RSKR queries. The proposed model to estimate the optimal grid cell size is discussed in Sect. 5. The parallel implementation using the slicing technique appears in Sect. 6. Our algorithms and theoretical models are experimentally evaluated in Sect. 7, while conclusions and future work appear in Sect. 8.

2 Related work

2.1 Top-k spatial queries

There are several variants of top-k spatial queries studied extensively in literature. A top-k spatial query returns k objects according to various query-provided spatial properties. For example, top-k spatial preference queries return a ranked set of the k best objects based on the scores of objects in their spatial neighborhood [33,53–55]. The top-k spatial join query retrieves the k spatial objects from two sets that intersect the largest number of objects from the other set [32,58]. There are also several variations of top-k spatial queries that deal with similarity among trajectories [26,50]. Recent research focused on retrieving the k most frequent or trending keywords over query-provided spatio-temporal ranges. [27,35] address this query over a streamed dataset, while in [2] we considered a disk-resident archived dataset. We are different from these works in that we address reverse top-k queries where the keyword is a query parameter. A top-k spatial *keyword* query takes into account not only the objects' spatial properties, but also keywords provided in the query. For example, [5,12] consider the problem of finding k objects that are closest to the query location and contain the query keywords. Similar problems have also been studied in the context of spatially annotated web objects [4,9]. Our work is orthogonal as we return areas where the query keyword is among the top-k most frequent.

2.2 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 [1]. Other examples include RkNN for spatial–textual similar objects [24,25], decision support and identifying potential customers [21,37], Reverse Spatial and Textual k Nearest Neighbor (RSTkNN) query used for interested sets [8,56] and the Reverse top-k Boolean spatial keyword query [14]. While we also look at 'reverse' queries, we return regions instead of data objects.

Vlachou et al. [45] 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. [45,46,49] propose several threshold-based algorithms to solve reverse top-k queries, while [30] addresses parallel and distributed processing of the reverse top-k query. Reverse top-k queries can be used to identify the most influential products [48] or monitor the popularity of locations based on user mobility [47].

The Spatial Reverse Top-k query considers the spatial distance between the locations of users and facilities as one of the criteria [31,52]. The distance between a user u and a facility f depends on the locations of u and f. Consequently, the distance value of a facility is different for each user. PCK [31] can only handle $k = 1$ and two attribute/criterion (including the distance criterion). Yang et al. [52] present a region-based pruning algorithm that can handle arbitrary number of attributes and $k \geq 1$. Our work does not consider the distance between the posts but rather focuses on the keywords in the post and the location associated with the post.

More related to our work is the Reverse *spatial* Top-k Keyword 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 [11] 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 [20] 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 Sect. 7.5, including an experimental comparison.

2.3 Density and burstiness queries

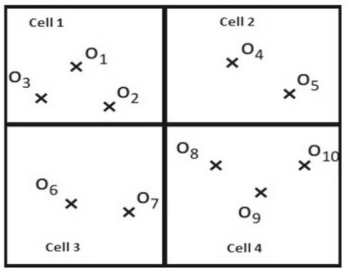
Related are also works on density-based queries over moving object databases. A spatial area is *dense* if the number of moving objects it contains is above some threshold [15]. Related queries include finding ROIs [34,41,42], convoys [19], flocks [44], assemblies [43], etc. While we consider density 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 [22]. [28] 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 [23], we extended the problem to identifying spatio-temporal 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 Problem definition

Let $\mathcal{D} = \{o_1, o_2, \ldots, o_N\}$ 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 $= \{t_1, t_2, \ldots\}$ 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, Fig. 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 a term t in R is $f_R(t) = \{\text{count}(o_i) | t \in o_i.\text{Terms} \ \& \ o_i.\text{Loc} \in R\}$. 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 paper, we assume the existence of a grid index I that will facilitate query answering (Fig. 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) pairs, sorted



(a)

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

(b)

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

(c)

Fig. 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

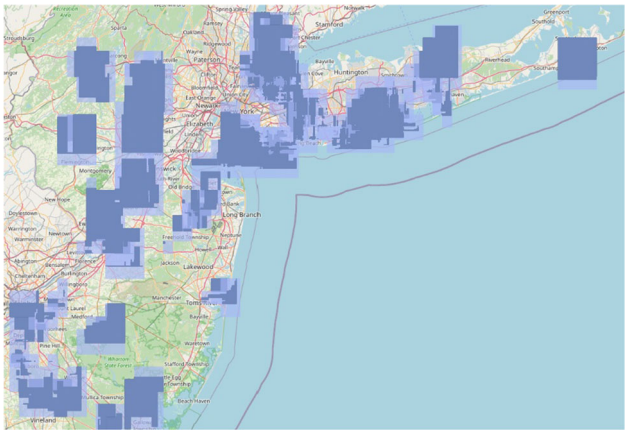


Fig. 2 RSK (deep blue) and RSKR (light blue) results for keyword "york" (colour figure online)

by decreasing frequency. A *pair (t,f)* of the STL indicates that that term t appears in f posts in that grid cell (Fig. 1c).

3.1 Reverse spatial top-k 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 and every point in these regions. Figure 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).

Computing the exact answer to the RSK problem involves checking many neighborhoods and is thus expensive. For that reason, we also propose a restricted version of the problem (RSKR) defined next. In particular, RSKR returns cells with at least one point where q is (k,l)-frequent.

3.2 RSK-restricted (RSKR) query

An RSKR query Q_R is again a tuple $\langle k, q, l \rangle$; however, the answer to 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 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 coincide 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.

The query choice (RSK vs RSKR) depends on the application's need in: (1) query speed and (2) spatial accuracy. Answering RSKR queries is swift (ms) compared to addressing RSK queries, as we will see in the experiments (Sect. 7). On the other hand, by definition, the RSKR query is less accurate as it only requires one answer per returned Grid cell (while in RSK, each point in the areas returned is an answer). We further discuss accuracy in Sect. 7.3.

3.3 System architecture

There are several works that study how to efficiently select posts based on various criteria like location [40], time [3,40], keywords [3,38,40], sentiment [55,57], or topic [38,40]. We rely on these works to select relevant posts that will be input to our algorithms to answer RSK and RSKR queries, as shown in Fig. 3. We assume that after the data is selected, it is indexed using the uniform GRID and building the STLs. Here we focus on the query execution time (that includes the filtering and refinement steps). The time to index the data, which corresponds to the Indexing module in Fig. 3, is negligible (around 1%) compared to the RSK algorithm execution, but not compared to the much faster RSKR algorithm, as we will see in Sect. 7.

Note that we do not need to re-execute the Data Selection and Indexing phases when using a different keyword as input. This is only needed when we change the selection criteria that pick the relevant posts for the input to the query (for example, only keep posts related to a topic or change the time or spatial range).

As long as the Data Selection criteria remain the same, we can reuse the STLs created for one query across multiple queries if l is at least $2c$ where c is the side length of the cells in the GRID (this condition is elaborated later in the paper). By choosing a small enough cell size, this case is highly likely to happen.

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 $O(\frac{l^2 N}{A})$,

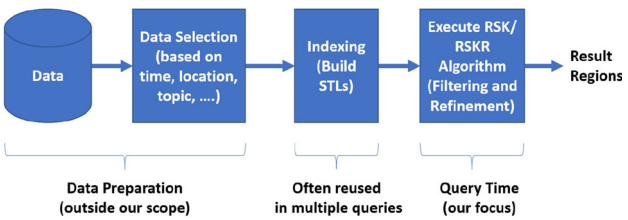


Fig. 3 System architecture to answer RSK and RSKR queries

resulting to $O(N^3)$ behavior (since area A and neighborhood size l are not dependent on N). Here, A is the area covered by the dataset, N is the number of posts, and l is the side length of the neighborhood. So, a square-shaped neighborhood with the area of l*l contains an average of $\frac{l^2 N}{A}$ posts. To determine whether the $l \times l$ neighborhood is an answer to the problem or not, we need to process the textual content of each of the posts at least once. Assuming processing textual content of a post is constant, we estimate the work for each $l \times l$ window to be $O(\frac{l^2 N}{A})$.

For both the RSK and RSKR queries, our proposed query processing algorithm consists of two steps: (a) **Filtering step**: This step aims to quickly prune the search space by separating the most promising (GREEN) and most unpromising areas (RED), hence the naming of the step. This step is critical in reducing query latency. Using the stored STLs, we identify the cells that are guaranteed to be in the answer (accepts) and those that are guaranteed not to be in the 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**: In this step, we take a closer look at the candidate cells to find out whether they are answers or not, i.e., refine the results. 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 Sect. 4.2, while the (several variants of the) refinement step for RSKR in Sect. 4.3.

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 (c) of the grid index I must follow $c \leq \frac{l_{\min}}{2}$. 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 $\eta_L = \lfloor \frac{l}{2c} \rfloor$. Inspired by the work in [29], we define the *conservative region* $\mathcal{CR}_{i,j}$ for a cell $C_{i,j}$ as the union of cells $C_{u,v}$ for which $i - \eta_L \leq u \leq i + \eta_L$ 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 $j - \eta_H \leq v \leq j + \eta_H$. Figure 4a and b shows 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 $\mathcal{CR}_{i,j}$, so p's l-square neighborhood completely contains $\mathcal{CR}_{i,j}$. Hence, the $\mathcal{CR}_{i,j}$ 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 $\mathcal{CR}_{i,j}$ is greater than the frequency of the kth 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 kth term in $\mathcal{CR}_{i,j}$, 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).

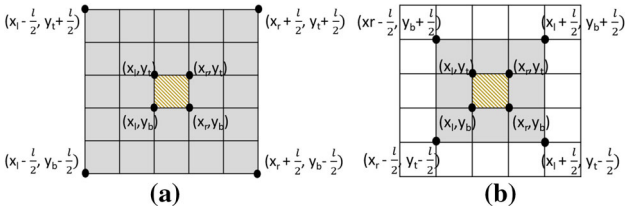


Fig. 4 **a** Cell expansive region for $\eta_H = 2$, **b** cell conservative region for $\eta_L = 1$

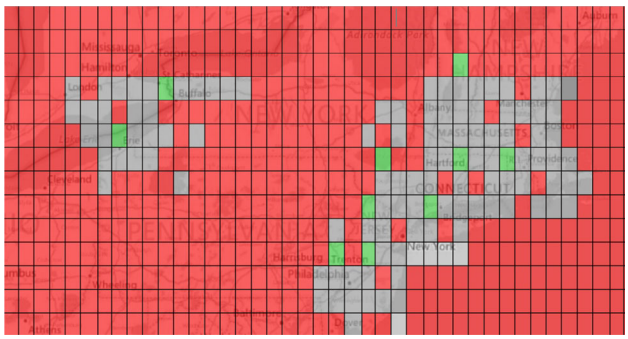
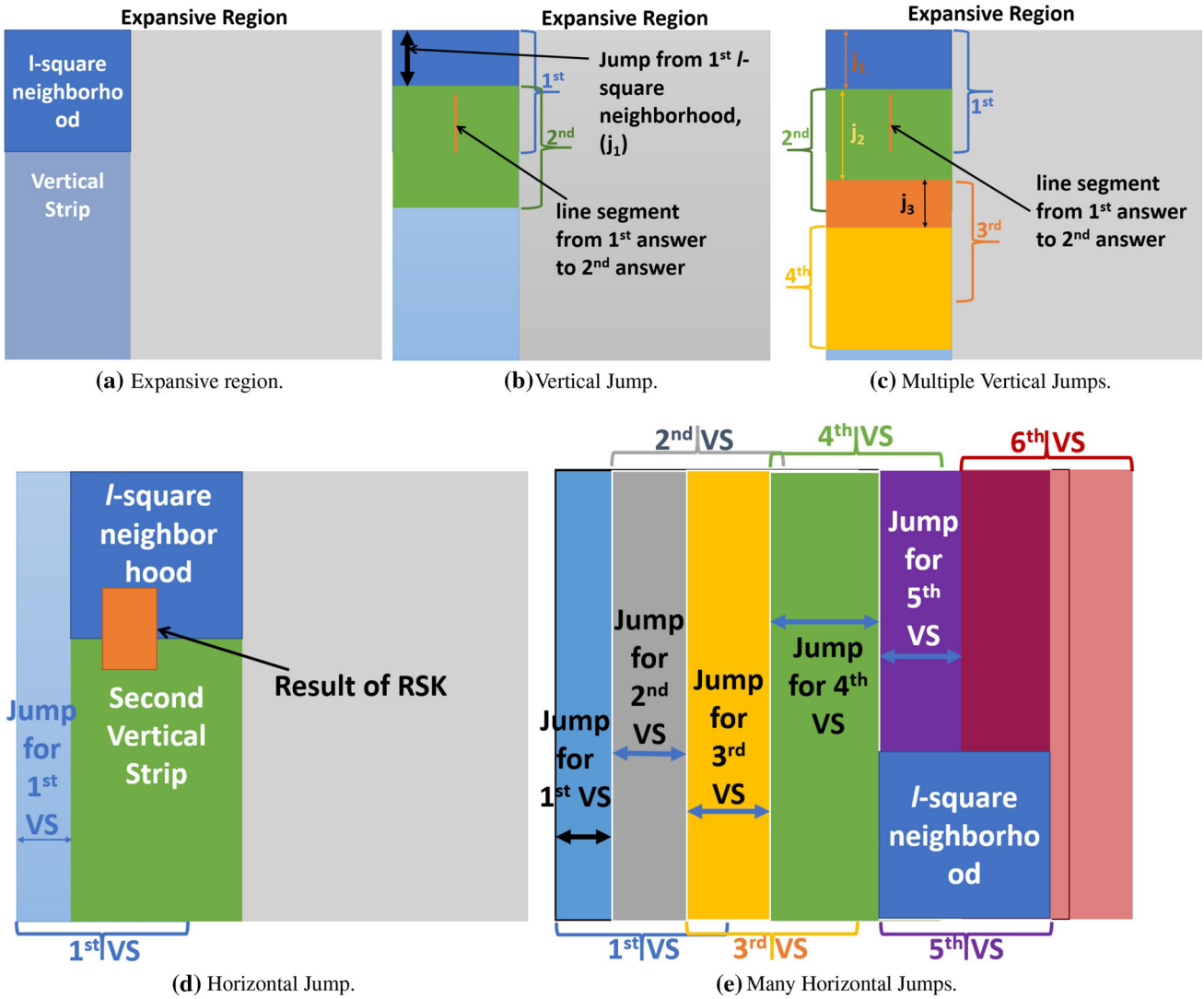


Fig. 5 Example output of the filtering step

The rest of the cells are candidate cells and we color them as GRAY. Only partial regions 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 5 shows an example output of the filtering step.

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 Fig. 4).



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

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

Fig. 6 Steps of the proposed algorithm for RSK query

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 paper, we discuss one way to traversing the XY-plane by initially fixing the region on X-axis and moving on the Y-axis. After we are done on 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, let us define some terms we will frequently use in the paper.

4.2.1 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 Fig. 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 1st l-square neighborhood, along the top border as shown in Fig. 6a.

It takes a lot of time to process the posts that are contained in the l-square neighborhood and add them in a hash table (we call it termFreqMap). 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 $\mathcal{CR}$ 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 $\mathcal{CR}$. After that, we process the posts in $\mathcal{P}$ to calculate the hashtable termFreqMap. Finally, we combine all the STLs from the cells in $\mathcal{CR}$ into termFreqMap. Then we fetch the frequency f_q of q and f_k of the kth most frequent keyword, from termFreqMap. We use the QuickSelect algorithm [16] to fetch the score for the kth 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 is 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.

4.2.2 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 o_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 kth 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**. In the 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 Fig. 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.

4.2.3 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 an 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 Fig. 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.

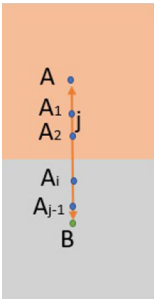


Fig. 7 Horizontal jump length estimation

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 Fig. 7. The point x between A and B (and also in the whole middle 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 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 1. Figure 6e shows an example where the 6th vertical strip is beyond the expansive region boundary (shown as a purple vertical line on the right), so the refinement step stops at the 5th vertical strip. Each horizontal jump stretches the line segment(s) generated in one vertical strip into rectangles (as shown in Fig. 6d). The width of these rectangles depends on the horizontal jump size. The orthogonal polygons in the RSK query result (Fig. 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 $O(\frac{l^2 N^3}{jA})$ 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.

Algorithm 1 $RSK(cell, q)$

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

```
posts ← getAdjacentposts(c)
sortpostsLongitude(posts)
while true do
    cell ← Cell(left, topBorder)
    currentPosts ← postsIn(posts, cell)
    sortpostsLatitude(currentPosts)
    top ← topBorder
    while true do
        cellY ← newCell(VerticalStrip, left, top)
        for each post in currentposts do
            currentPostsY ← post
            if not in previousGrid then
                newlyAddedpost ← post
        if prevGridexists then
            termFreqMap ← processRemovedposts()
            termFreqMap ← processNewlyAddedposts()
        else
            termFreqMap ← processposts(currentPostsY)
        q_score ← termFrequencyMap.get(q)
        k^th Score ← quickSelect(termFreqMap.values(), k)
        if q_score ≥ k^th Score then
            result ← cellY
            c.color ← GREEN
        jump ← |k^th Score - q_score|
        jumps ← add(jump)
        prevGrid ← cellY
        if cell.bottom ≥ bottomBorder then
            break
    minJump ← min(jumps)
    VerticalStrip ← jump(VerticalStrip, minJump)
    if cell.right ≥ rightBorder then
        break
return result
```

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).

4.3.1 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 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 kth 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 update the previous termFreqMap only when there is enough overlap (at least 50%) between subsequent l-square neighborhoods; otherwise, it will calculate a new termFreqMap from scratch. 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 2.

Algorithm 2 $RSKR - Approximate(GRID, q)$

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 ← calculateRandomPoint(cell)
4:     topLeftCell ← calculateTopLeftCell(cell)
5:     scoreTopLeft ← calculateScore(topLeftCell)
6:     if scoreTopLeft > 0 then
7:       cell.color ← GREEN
8:       break
9:     topRightCell ← calculateTopRightCell(cell)
10:    scoreTopRight ← calculateScore(topRightCell)
11:    if scoreTopRight > 0 then
12:      cell.color ← GREEN
13:      break
14:    bottomLeftCell ← calculateBottomLeftCell(cell)
15:    scoreBottomLeft ← calculateScore(bottomLeftCell)
16:    if scoreBottomLeft > 0 then
17:      cell.color ← GREEN
18:      break
19:    bottomRightCell ← calculateBottomRightCell(cell)
20:    scoreBottomRight ← calculateScore(bottomRightCell)

21:    if scoreBottomRight > 0 then
22:      cell.color ← GREEN
23:      break
24:    max ← max(scoreTL, scoreTR, scoreBL, scoreBR)
25:    if max ≤ score then
26:      break
27:    if max == scoreForTopLeft then
28:      rightBorder ← randomX
29:      bottomBorder ← randomY
30:    if max == scoreForTopRight then
31:      leftBorder ← randomX
32:      bottomBorder ← randomY
33:    if max == scoreForBottomLeft then
34:      rightBorder ← randomX
35:      topBorder ← randomY
36:    if max == scoreForBottomRight then
37:      leftBorder ← randomX
38:      topBorder ← randomY
39:    else
40:      break
41: return result
```

We implemented two additional 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 multiply its scores by the percentage of overlap under the assumption that the posts are uniformly distributed. (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 Fig. 2).

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 Sects. 4.2 and 4.3. Note that the cell size can be fixed in advance. Using the optimal cell size for indexing the data will result in better query latency for the RSK and RSKR problems. However, given any user-chosen cell size c, the algorithms presented will still support a query as long as $l \geq 2c$. 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 1 summarizes the notation used in the analysis.

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 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.

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

Proof As illustrated in Fig. 8, there are at most two partially overlapping cells along each dimension, i.e., one partial cell

Table 1 Notations used throughout theoretical analysis

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 l-square neighborhood
ρ	Average number of posts per cell
y	Average number of terms per post
K	First parameter in Heap's law [10]
β	Second parameter in Heap's law
I	Number of cells fully contained in 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

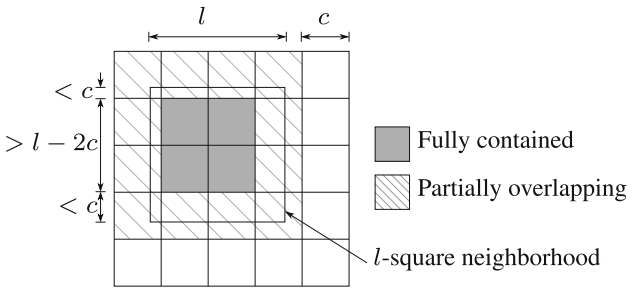


Fig. 8 Fully contained cells and partially intersecting cells in the l-square neighborhood

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 are at least $\lceil l/c \rceil - 1$. This means that there are 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 [10], $STL_{size} = K(\rho \cdot y)^{\beta}$, 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,

$$\begin{aligned} T_I &= I \cdot \mathrm{STL}_{\mathrm{size}} \\ &= \left(\frac{l}{c} - 1\right)^2 \cdot K(\rho \cdot y)^{\beta} \\ &= \left(\frac{l}{c} - 1\right)^2 \cdot K\left(\frac{c^2 N y}{A}\right)^{\beta} \end{aligned}$$

5.3 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 are at most* $4\frac{l}{c}$ *partially intersecting cells in the* l*-square neighborhood.*

Proof According to Lemma 2, there are at least $\frac{l}{c} - 1$ fully contained cells along each of the two dimensions. Additionally, there are 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 $(\frac{l}{c} - 1)^2 \cdot c^2$, thus, the area covered by partial cells is $l^2 - (\frac{l}{c} - 1)^2 \cdot c^2 = 2lc - c^2$. 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 Eq. 1.

$$\begin{aligned} \theta(c) = &\left(\frac{l}{c} - 1\right)^2 \cdot K\left(\frac{c^2 y N}{A}\right)^{\beta} + \frac{4lcN}{A} \\ &+ \frac{yN}{A}(2lc - c^2) \end{aligned} \tag{1}$$

The optimal cell c_* is the cell size that minimizes the value of θ in Eq. 1; to find c_* we use Wolfram Alpha [51].

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 $\frac{B}{G} \cdot \theta(c)$ 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 $\beta = 0.07197$. Equation 1 has a local minimum at $c = 0.0346488$ and local maximum at $c = 0.840214$.

5.4 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 Sect. 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 Sect. 5.1, and the processing cost is given by Eq. 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 kth keyword, $|f_q - f_k|$ in the l-square neighborhood, $j = \left|\frac{l^2 c^2 y N}{A} \cdot \left(\frac{1}{r_q} - \frac{1}{k}\right)\right|$, 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, $l_{\text{Count}_{\text{VS}}} = \left(N_{\text{VS}} - \frac{N \cdot l^2}{A}\right) / j$. From steps 2–4, the cost of processing one vertical strip is, $\text{cost}_{\text{VS}} = \text{sort}_{\text{VS}} + \theta + j \cdot y \cdot l_{\text{Count}_{\text{VS}}}$.
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, $\text{VS}_{\text{Count}_E} = \left\lceil \frac{2 \cdot (N_E - N_{\text{VS}})}{j} \right\rceil$.

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

$$\begin{aligned}\theta_F(c) &= \text{sort}_E + \text{cost}_{VS} \\ &= \text{sort}_E + (\text{sort}_{\text{VS}} + \theta + jy \cdot l_{\text{Count}_{\text{VS}}}) \cdot \text{VS}_{\text{Count}_E} \\ &= N_E \log(N_E) \\ &\quad + (N_{\text{VS}} \log(N_{\text{VS}}) + \theta + jy \cdot l_{\text{Count}_{\text{VS}}}) \cdot \text{VS}_{\text{Count}_E}\end{aligned}$$

The optimal cell c_* is the cell size that minimizes $\theta_F(c)$ as before, we find c_* using Wolfram Alpha [51].

6 Parallel implementation

6.1 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. Our proposed technique assumes a shared-nothing configuration which makes it highly portable to many distributed processing systems such as Hadoop or Spark. During the RSK refinement step, the time taken by each candidate cell varies substantially (see Fig. 9). 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.

6.2 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*

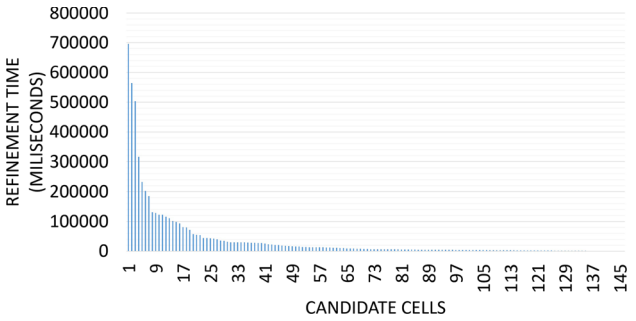


Fig. 9 Refinement time for different candidate cells for q = "home" and $k = 5$

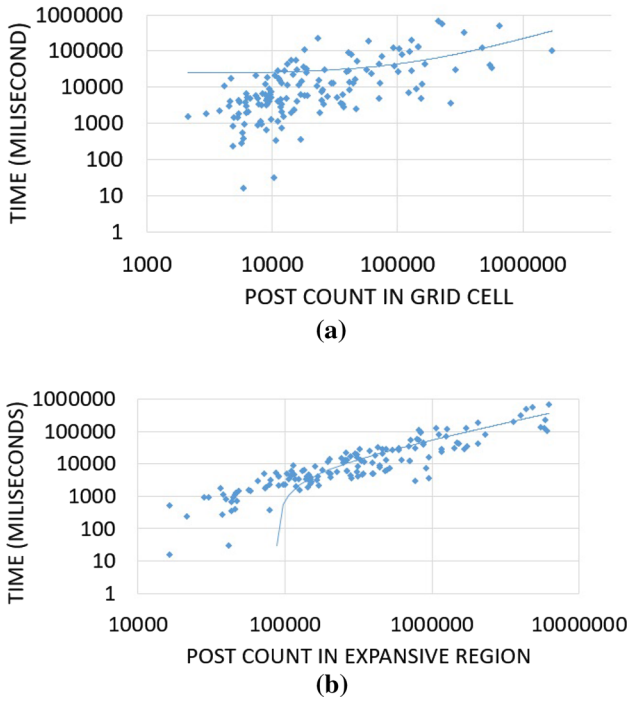


Fig. 10 Correlation between refinement time and cell post count (left), expansive region post count (right), for q = "home", $k = 5$

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; Fig. 10 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

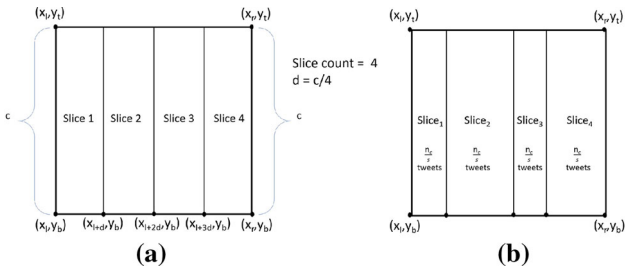


Fig. 11 Equal-width (left) and equal-post slicing (right)

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.

6.3 Workload distribution

Our next step is to choose an efficient workload distribution strategy to balance the workload of each cluster node. The naïve way to do this would be to distribute the candidate cells among the nodes randomly so that each node gets similar number of candidate cells to process. This approach is not very efficient as we show in Sect. 7.4. The reason is that the refinement time for each candidate cell varies widely (Fig. 9), as the refinement time is highly correlated with the posts count in the expansive region. Instead, we distribute the candidate cells among the cluster nodes in such a way that each node may have different number of candidate cells but has similar number of posts to process in the refinement step (i.e., based on the sum of the total number of posts in the expansive regions of the candidate cells assigned to that node).

6.4 Slicing

Even with a perfect load balance, the running time of the parallel algorithm will be bounded by the most expensive cell. In this part, we propose how to increase the parallelization granularity and reduce the upper-bound of the cell computation time by *slicing* candidate cells, so that expensive cells can be refined in parallel.

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 $\gamma \cdot \mathcal{M}$ 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 $\tau = \frac{\sum_G N_E}{\mathcal{M} \cdot \gamma}$ 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 11 (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 11 (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 ($\frac{\text{jump}_{\min}}{2}$) and thus is typically smaller than a vertical jump.

6.5 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).

7 Experiments

7.1 Setup

7.1.1 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 32 GB of memory. We run one executor per core (vCPU).

7.1.2 Datasets

Using the Twitter streaming API [39], 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 6-month period from August 2014 to January 2015. This resulted in a dataset with 15M geo-tagged tweets (12 GB in size). In particular, since most users do not typically reveal their phone's GPS [6,18], 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 stop-words, each tweet has an average of 5 terms.

In addition to the Twitter dataset, we also used the Chicago crime dataset [7] for some experiments. This dataset reflects reported incidents of crime that occurred in the City of Chicago from 2001 to February 11th, 2022. This dataset has 7M crime reports from Chicago and surrounding areas. Every crime report contains point location, time, crime type, and textual description. It is 4.1 GB in size and contains the following fields: *Crime Report Type, Latitude, Longitude*, and so on. We conducted experiments with the location (latitude and longitude) and crime report type field, which will give us

Table 2 Keyword ranks, Twitter dataset (left), Crime dataset (right)

Symbol	Rank	Keyword	Symbol	Rank	Keyword
Q_1	1	Love	Q_1	1	Theft
Q_5	5	Good	Q_2	2	Battery
Q_{10}	10	Sorry	Q_5	5	Narcotics
Q_{25}	25	Home	Q_7	7	Assault
Q_{50}	50	Hope	Q_{10}	10	Vehicle
Q_{200}	200	Change	Q_{15}	15	Trespass

areas where a particular crime is among the top-k most frequent crimes (that are reported). Each crime report contains around two terms on average, as most crime types are short. Below most experiments use the Twitter dataset (by default); when the crime dataset is used it is specifically mentioned.

As discussed in Sect. 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 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 28.5 K tweets with GPS location per day for the above rectangle A. For all our experiments, we assume that the dataset size fits in main memory. This is a realistic assumption as our picked dataset of 15 million tweets covers around 6 months for the larger New York State and fits in the main memory of our cluster.

7.1.3 Query keywords

In our experiments, we use query keywords that have different ranks in the dataset based on their frequencies. Table 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 $k = 10$. For the Chicago crime dataset, we use keywords with ranks 1, 2, 5, 7, 10, 15 based on the frequency in the full dataset, as shown in Table 2. Given the limited types of crime, the vocabulary for this dataset is small. That is why the query keywords' ranks are chosen as such; for the same reason, we chose the value of $k = 3$.

7.1.4 Index structure

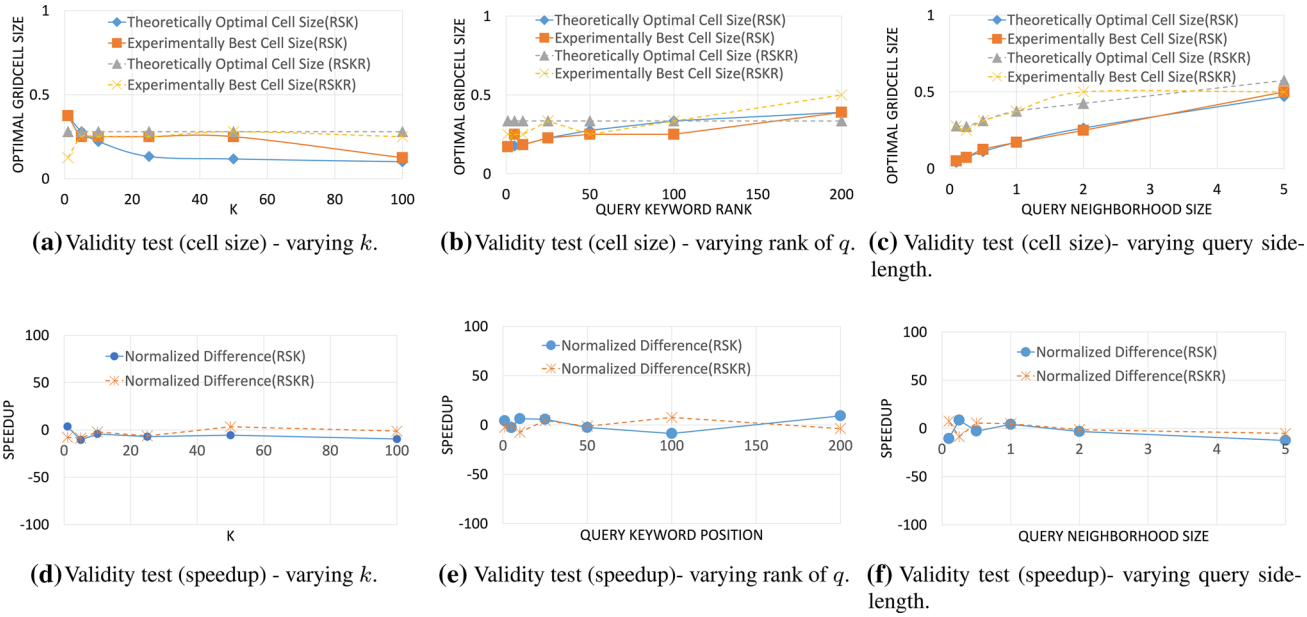
We divided the space covered by each dataset into a uniform grid of square cells with each cell containing tweets that are in the geographical area covered by that cell. Each cell is enhanced with a STL of it's tweets.

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.

The validity results with respect to cell size for the RSK and RSKR problem appear in Fig. 12a–c. In these experiments, we used the RSK algorithm with vertical and horizontal jumps (Algorithm 1) 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 does not show any abrupt changes. Figure 12d–f presents 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 speedup: $\frac{\text{Latency}(c_\theta) - \text{Latency}(c_{\text{exp}})}{\text{Latency}(c_\theta)}$. 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 2 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 paper for the Twitter dataset, 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). For the crime dataset, we use cell size $c = 0.05$ and $l = 0.1$ because the area covered by the dataset is relatively smaller compared to the area covered by the Twitter dataset.



(a) Validity test (cell size) - varying k. **(b)** Validity test (cell size) - varying rank of q. **(c)** Validity test (cell size)- varying query side-length.

(d) Validity test (speedup) - varying k. **(e)** Validity test (speedup)- varying rank of q. **(f)** Validity test (speedup)- varying query side-length.

Fig. 12 Validity test with respect to cell sizes and speedup for cell size estimator

7.3 Single-node evaluation

We first examine the performance of the **RSK query**. The aim of these experiments is to analyze the query latencies among the different approaches used to answer 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**), RSK with vertical and horizontal jump without filtering step (**RSK-no Filter**), and RSK with both vertical and horizontal jump (**RSK-VH**, described in Algorithm 1). Figures 13 and 14 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. Figures 15 and 16 present the same experiment results for the crime database with varied query keyword positions q (using $l = 0.1$) and the query neighborhood size l (using keyword Q_5), respectively. The baseline approach (RSK-W) is too slow (took almost an hour) and hence is omitted from the figures. In all of the 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. RSK-no Filter performs better than RSK-V, but because of lacking the filtering step to prune its search space, RSK-VH is around 1.5–7 times faster than RSK-no Filter for different keywords. The more popular and less popular terms benefit the most from the filtering step.

We also observed in Fig. 13 that the query latency spikes for keywords whose rank is closer to k (in these experiments

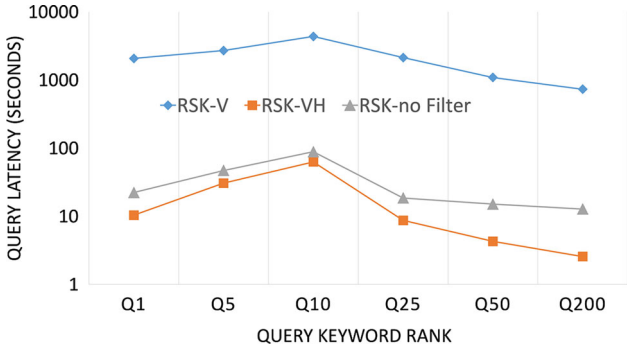


Fig. 13 RSK query latency for varying query keyword (Twitter dataset)

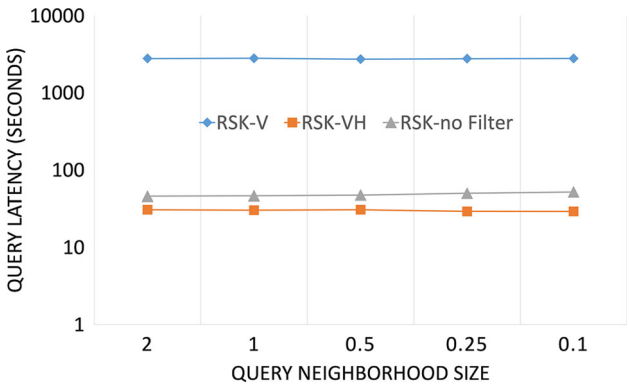


Fig. 14 RSK query latency for varying neighborhood sizes (Twitter dataset)

Q_{10}). When q is close to the kth 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

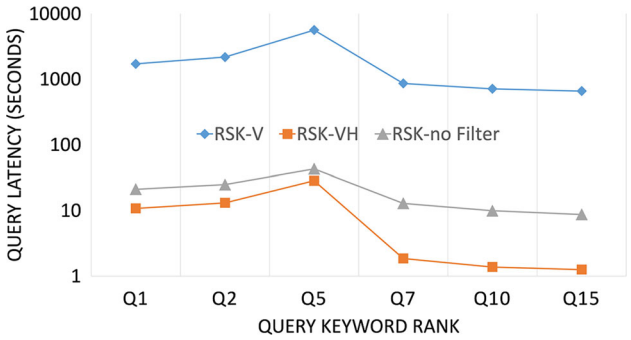


Fig. 15 RSK query latency for varying query keyword (crime dataset)

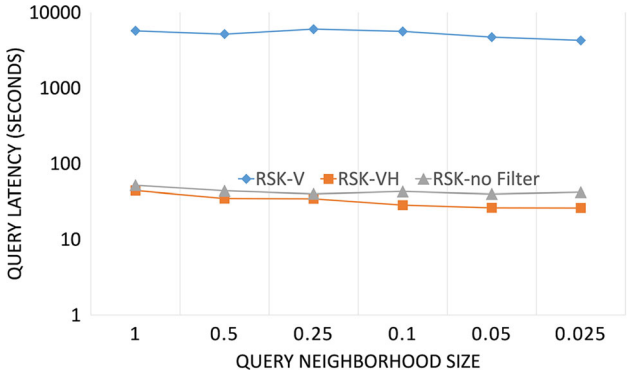


Fig. 16 RSK query latency for varying neighborhood sizes (crime dataset)

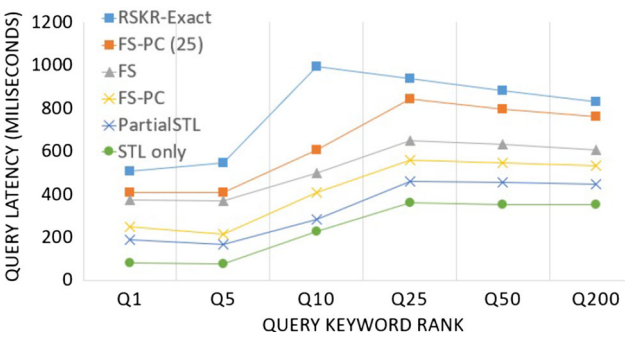


Fig. 17 RSKR query latency for varying query keyword (Twitter dataset)

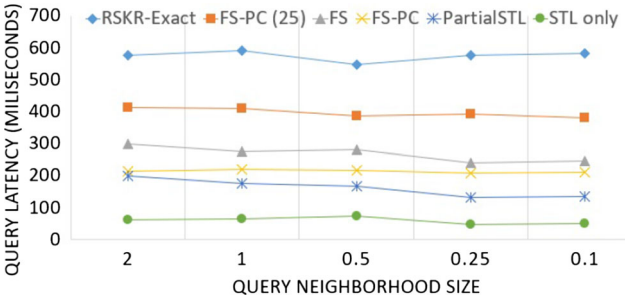


Fig. 18 RSKR query latency for varying neighborhood sizes (Twitter dataset)

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 1) for the (spatial) RSK query.

The next experiments consider the **RSKR query** performance using the same datasets (random sample with 10K tweets). The goal of these experiments is to (i) analyze the query latencies of the solutions to answer RSKR query, (ii) analyze the accuracy of the solutions. Unless otherwise specified, both number of random restarts and number of divisions for each algorithm are set to 5. As in each coordinate division, we check 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 17 and 18 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, for the Twitter dataset. Figures 19 and 20 present the same experiment results for the crime database with varied query keyword positions q (using $l = 0.1$) and the query neighborhood size l (using keyword Q_5), respectively. In all four 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 ms) 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

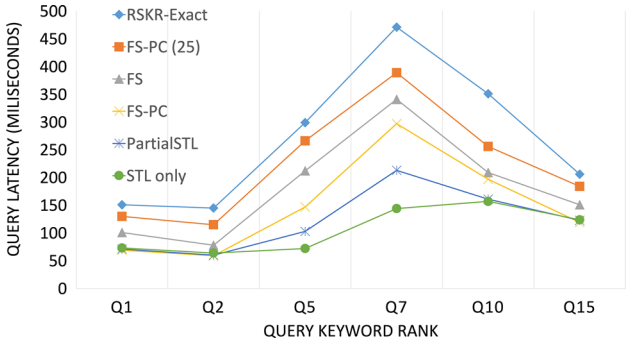


Fig. 19 RSKR query latency for varying query keyword (crime dataset)

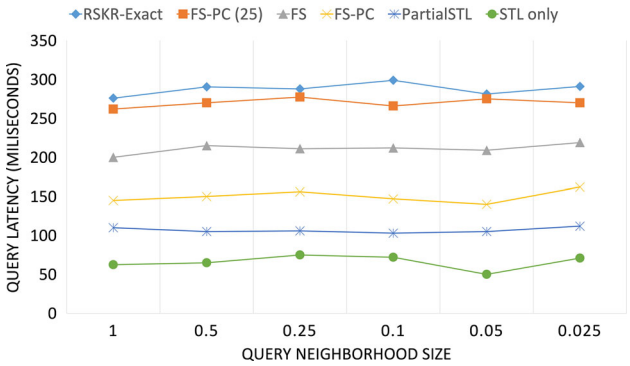


Fig. 20 RSKR query latency for varying neighborhood sizes (crime dataset)

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 Figs. 23 and 24) as it only considers the cells that are fully contained in the l-square neighborhood, completely ignoring the partially contained cells.

Another observation from Fig. 17 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 21 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. This figure also shows the actual effectiveness of our filtering step. We can see that the filtering step prunes a substantial portion of the total search space (77–99%), as all cells have the same area. In Figs. 22 (RSK) and 23 (RSKR), we will see that the time taken for filtering is negligible compared to the time taken by the refinement step. This filtering step plays a crucial role in reducing the overall query latency for RSK and RSKR queries, as shown in Figs. 13, 14, 15, and 16 from the difference of query latency between RSK-VH and RSK-no Filter.

Figure 22 shows the (average) number of l-square neighborhoods 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 neighborhoods to find an answer. The difference becomes more apparent in larger ranks since there are much fewer answers. Figure 22 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 Fig. 21 with the l-square neighborhoods checked per cell. This graph behaves similarly with the behavior seen in Fig. 17. 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 decreases 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 Figs. 23 and 24, respectively. While faster, the approaches that find results by looking only within STLs (namely the PartialSTL and STLOnly) suffer both in precision and recall 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

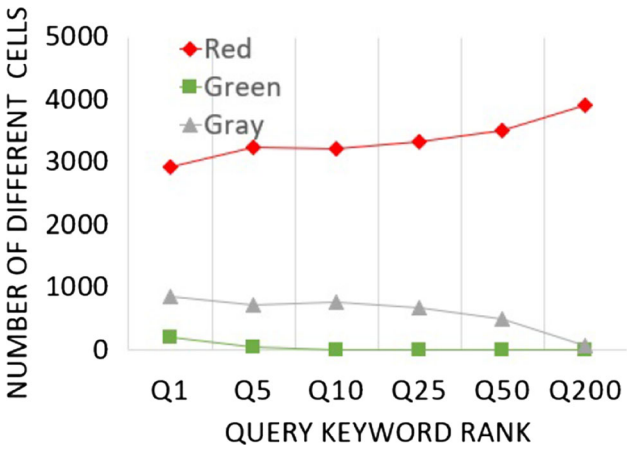


Fig. 21 Different cell counts for different query keywords

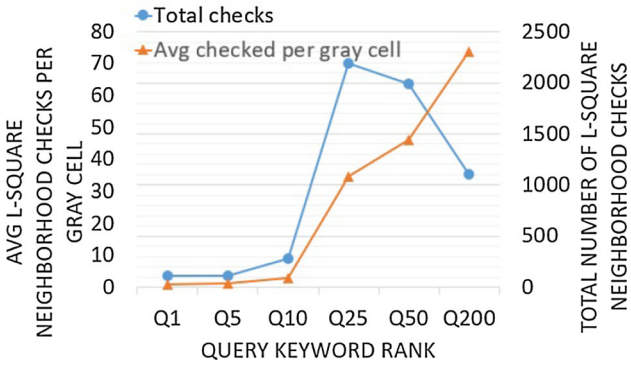


Fig. 22 Number of checked l-square neighborhoods for different query keywords

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 2).

7.3.1 Using the RSKR algorithm to answer a RSK query

Figure 25 shows the total area (sum of all rectangles returned as results) when using the RSKR (FS-PC) and the RSK-VH algorithms to answer various RSK queries. The total area returned by the RSKR algorithm is approximately between 1.5 times (for query Q_1) and 9 times (for query Q_{200}) more than the total area returned by the RSK algorithm. Figure 26 shows the accuracy when using the RSKR algorithm for RSK queries. We calculate the accuracy as: $\frac{\text{RSK}_{\text{Area}}}{\text{RSKR}_{\text{Area}}}$. The accuracy varies from 65% (for query Q_1) to 10% (for query Q_{200}).

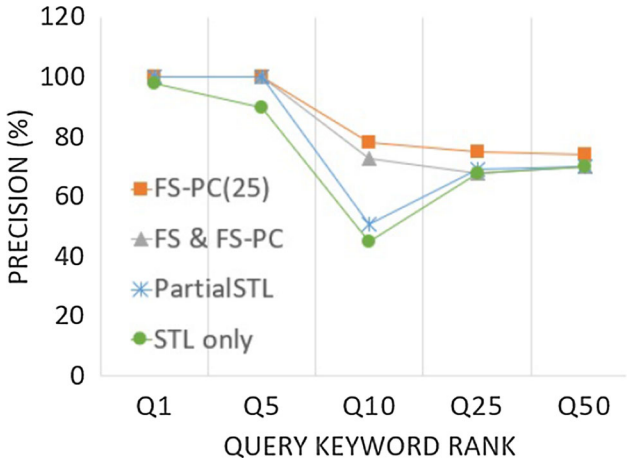


Fig. 23 Precision of RSKR approximate algorithms for different query keywords

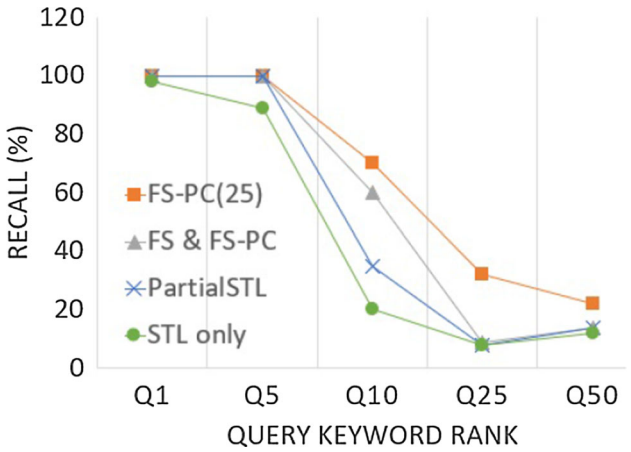


Fig. 24 Recall of RSKR approximate algorithms for different query keywords

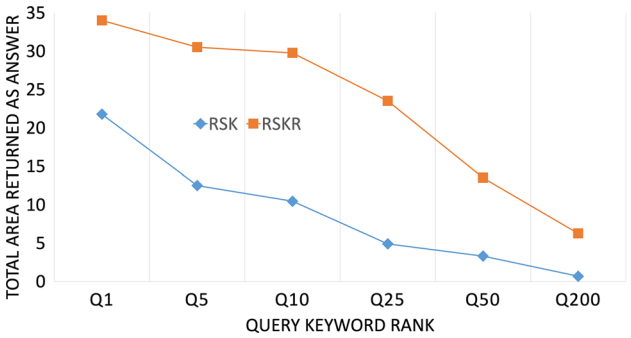


Fig. 25 Total area returned as answer by RSK and RSKR queries

7.3.2 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 27 and 28 show the query latency for each problem, respectively, for different keywords while

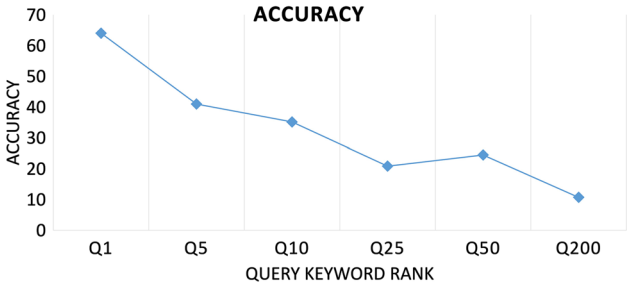


Fig. 26 Accuracy of RSKR algorithm when answering RSK queries

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.

7.3.3 Index time versus query time

As discussed in Sect. 3 (see system architecture in Fig. 3), this paper assumes that the indexing is done beforehand and only considers the filtering and refinement steps as part of the query time. Here, we explore how significant the indexing time is when compared to the algorithms' execution time (filtering and refinement). Figures 29 and 30 show the times for the indexing, filtering, and refinement steps to answer RSK and RSKR queries, respectively. As the size of the dataset increases, so does the time required to finish each step. We also notice that for the RSK, both the filtering and indexing steps take time that is negligible (around 1%) compared to the refinement step. Note that the indexing and filtering steps are the same for both the RSK or RSKR queries. On the other hand, the refinement step of RSKR is typically faster than the indexing step. This is because RSKR checks *much fewer* l-square neighborhoods than RSK.

7.4 Multi-node evaluation

7.4.1 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 multi-node experiments below

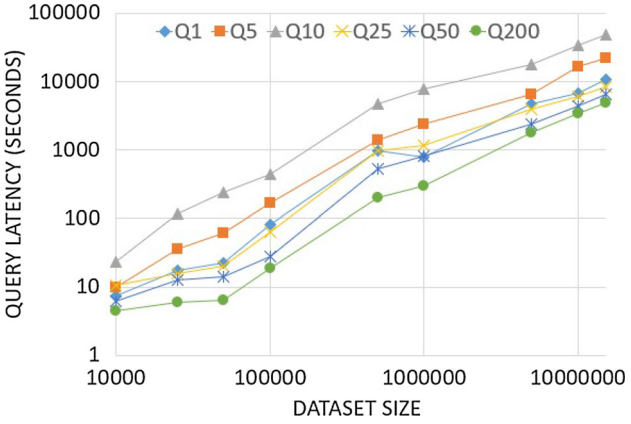


Fig. 27 RSK query latency while varying the dataset size

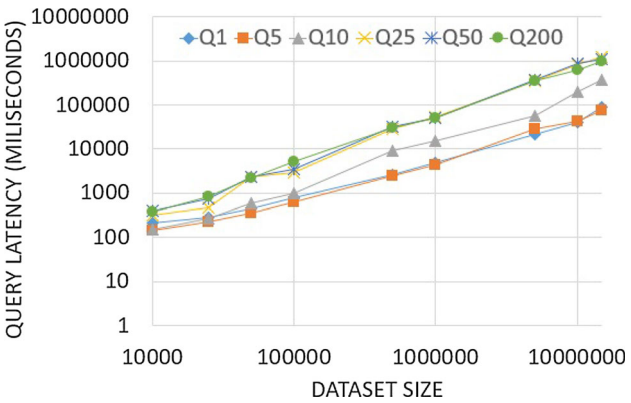


Fig. 28 RSKR query latency while varying the dataset size

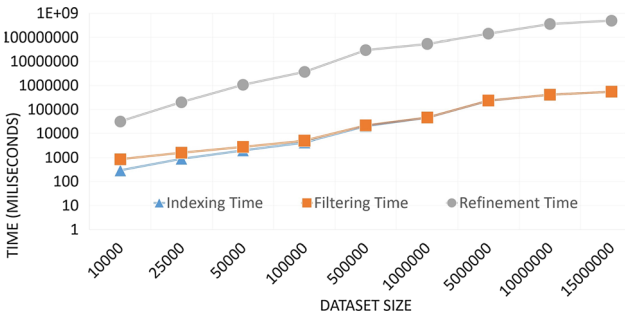


Fig. 29 Time comparison between Indexing, Filtering, and Refinement steps to execute the RSK query

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 and workload balancing, we experimentally evaluated three versions of the RSK algorithm; (i) RSK using straightforward random distribution (*Not Sliced - Random*), (ii) RSK using post count-based distribution (*Not Sliced*), and (iii) RSK with slicing using post count-based distribution (*Sliced*), where we applied equal-post slicing using the approach discussed in Sect. 6. The results appear in Fig. 31 for different query keyword positions (q). We observe

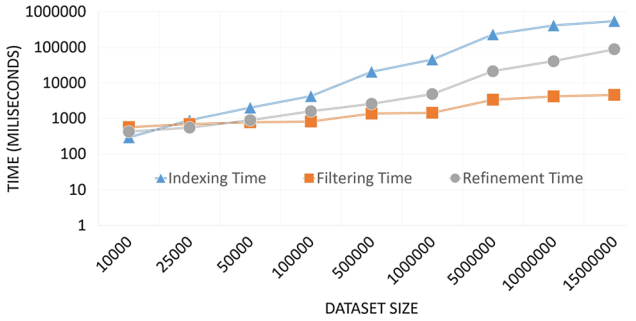


Fig. 30 Time comparison between Indexing, Filtering, and Refinement steps to execute the RSKR query

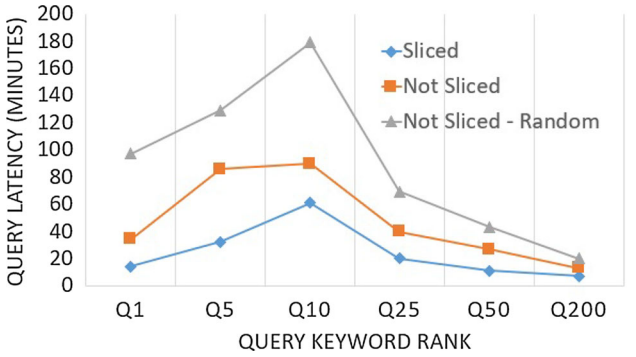


Fig. 31 Comparing effect of slicing on query latency (in minutes) of RSK for different query keywords

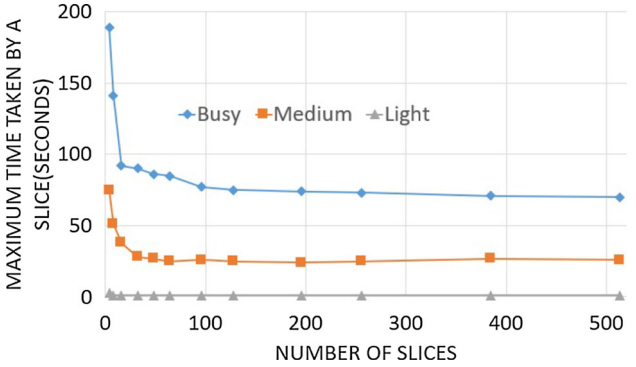


Fig. 32 Max. time for a slice for Q_1

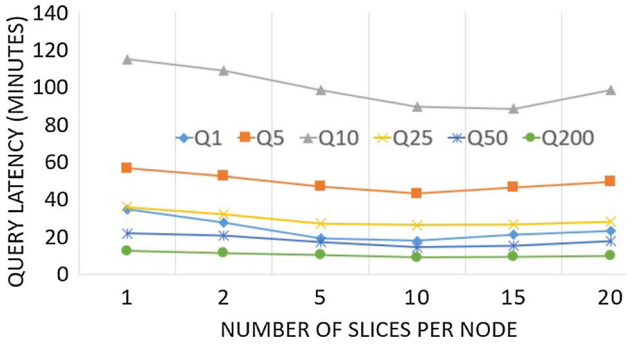


Fig. 33 Average slice count per core (γ)

significant query latency speedup if we distribute the candidate cells among the nodes by balancing the total post count in the expansive regions in each node compared to the random distribution. This is because the refinement time has high correlation with the post count in the expansive region (N_E) as discussed in Sect. 6 and the refinement time accounts for almost 99% of the total query latency of RSK query (Fig. 29). We also observe significant improvement in query latency by using slicing. This is because slicing provides even better workload balancing among the nodes.

Since individual slices can be processed at different cores, the maximum time taken to process any of the slices is important. Figure 32 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 by medium and light) cells as it provides a larger reduction in latency. This asserts the findings of Fig. 10 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 33 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 does not help. In the remaining experiments, we set $\gamma = 10$ as it depicted the best query latency.

7.4.2 Cluster speedup

To measure the speedup performance we started with a cluster of 50 cores doubling them until 400 cores. Figure 34 shows the RSK query latency (using Algorithm 1) 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

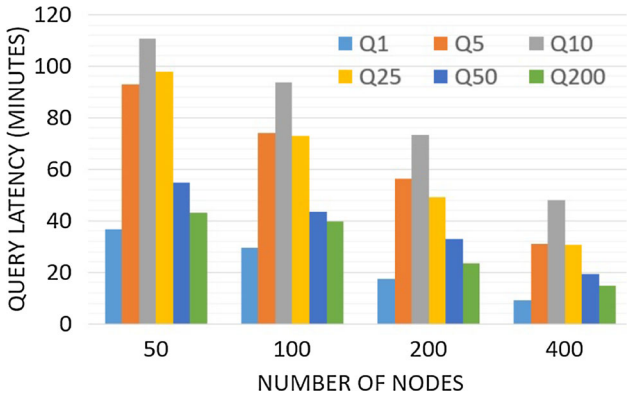


Fig. 34 RSK query latency with varying number of cores for different keywords

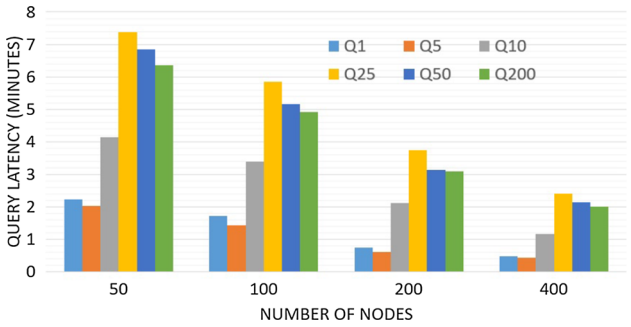


Fig. 35 RSKR query latency with varying number of cores for different keywords

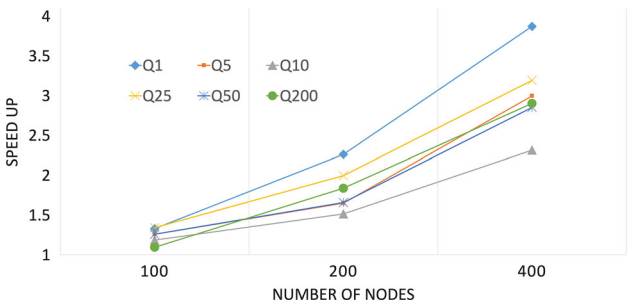


Fig. 36 Speedup of RSK algorithm with varying number of cores for different keywords

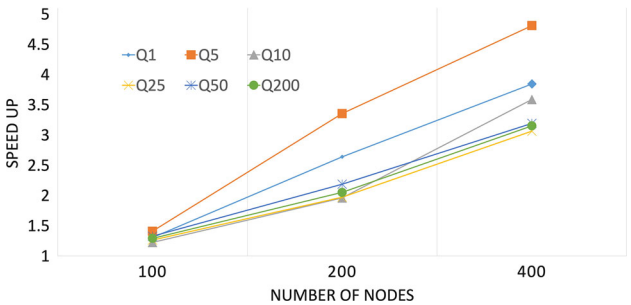


Fig. 37 Speedup of RSKR algorithm with varying number of cores for different keywords

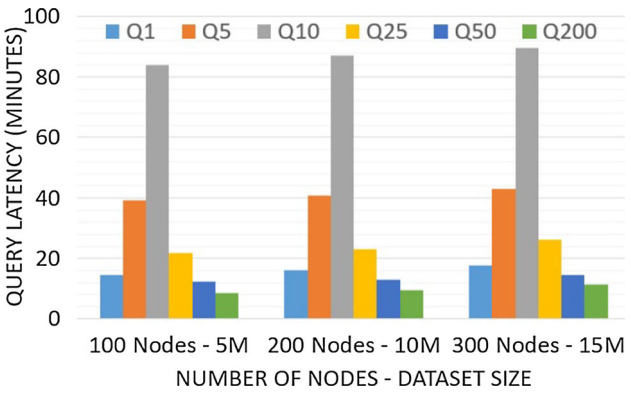


Fig. 38 RSK scale-up experiments for different keywords

with the single-node experiments (Fig. 13). Figure 35 shows query latencies for RSKR (Algorithm 2) 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 (Fig. 17).

The actual speedup is shown in Figs. 36 and 37 for RSK and RSKR, respectively. We calculated speedup as $S = \frac{L(50)}{L(p)}$. Here, $L(p)$ is the query latency with p cores; we used 50 cores as the baseline. The speedup for both parallel algorithms is increasing with the number of cores. It is however not proportional to the increase in the number of cores as the query latency depends on the slowest core to finish. While we distribute slices across cores to create equal loads, some slices can take much longer than others (as was seen in Fig. 32) thus affecting load balancing.

7.4.3 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. Figures 38 and 39 show query latency of RSK and RSKR (for different keywords) for the above scale-up experiments, respectively. 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.

7.5 Comparison with GARNET

GARNET [20] is the closest work to ours; however, it has two key limitations compared to our approach. First, the neighborhood size in GARNET has to be equal to the cell size. Second, all the results have to be aligned with the cell of the

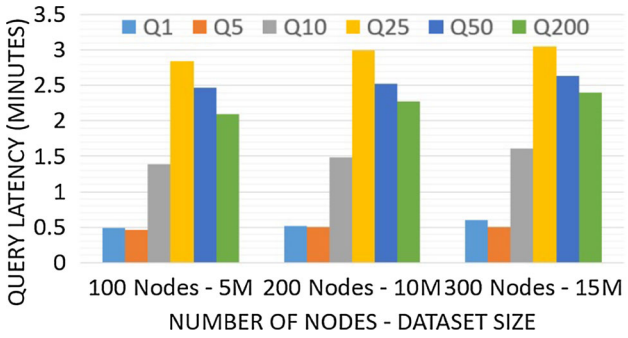


Fig. 39 RSKR scale-up experiments for different keywords

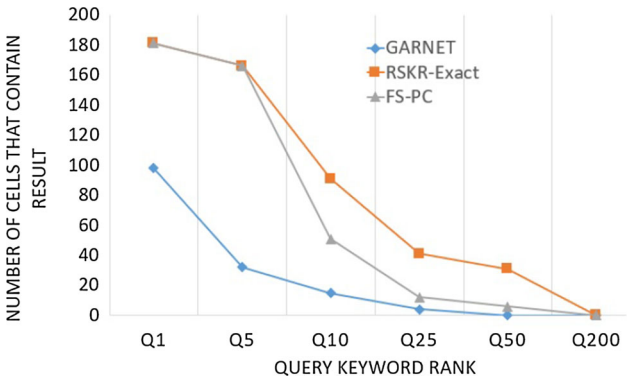


Fig. 40 Comparison of number of results found by GARNET, RSK-Exact, and FS-PC algorithm

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 40 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 numerous shifted l-square neighborhoods within each cell.

8 Conclusions and future work

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. An interesting future direction is to explore RSK-related queries over spatio-temporal data.

Acknowledgements This work was partially supported by NSF grants SES-1831615, IIS-1954644, IIS-1901379 and IIS-1838222.

References

1. Achtert, E., Böhm, C., Kröger, P., Kunath, P., Pryakhin, A., Renz, M.: Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In: Chaudhuri, S., Hristidis, V., Polyzotis, N. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27–29, 2006, pp. 515–526. ACM (2006). https://doi.org/10.1145/1142473.1142531
2. Ahmed, P., Hasan, M., Kashyap, A., Hristidis, V., Tsotras, V.J.: Efficient computation of top-k frequent terms over spatio-temporal ranges. In: Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., Suciu, D. (eds.) Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14–19, 2017, pp. 1227–1241. ACM (2017). https://doi.org/10.1145/3035918.3064032
3. Busch, M., Gade, K., Larson, B., Lok, P., Luckenbill, S., Lin, J.J.: Earlybird: Real-time search at twitter. In: Kementsietsidis, A., Salles, M.A.V. (eds.) IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1–5 April, 2012, pp. 1360–1369. IEEE Computer Society (2012). https://doi.org/10.1109/ICDE.2012.149
4. Cao, X., Cong, G., Jensen, C.S.: Retrieving top-k prestige-based relevant spatial web objects. Proc. VLDB Endow. **3**(1), 373–384 (2010). https://doi.org/10.14778/1920841.1920891
5. Chen, L., Cong, G., Jensen, C.S., Wu, D.: Spatial keyword query processing: An experimental evaluation. Proc. VLDB Endow. **6**(3), 217–228 (2013). https://doi.org/10.14778/2535569.2448955
6. Cheng, Z., Caverlee, J., Lee, K.: You are where you tweet: a content-based approach to geo-locating twitter users. In: Huang, J., Koudas, N., Jones, G.J.F., Wu, X., Collins-Thompson, K., An, A. (eds.) Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26–30, 2010, pp. 759–768. ACM (2010). https://doi.org/10.1145/1871437.1871535
7. Chicago crime dataset, https://star.cs.ucr.edu/?Chicago%20Crimes#center=42.013,-86.749&zoom=9
8. Choudhury, F.M., Culpepper, J.S., Sellis, T., Cao, X.: Maximizing bichromatic reverse spatial and textual k nearest neighbor queries. Proc. VLDB Endow. **9**(6), 456–467 (2016). https://doi.org/10.14778/2904121.2904122
9. Cong, G., Jensen, C.S., Wu, D.: Efficient retrieval of the top-k most relevant spatial web objects. Proc. VLDB Endow. **2**(1), 337–348 (2009). https://doi.org/10.14778/1687627.1687666
10. Egghe, L.: Untangling Herdan's law and Heaps' law: mathematical and informetric arguments. J. Assoc. Inf. Sci. Technol. **58**(5), 702–709 (2007). https://doi.org/10.1002/asi.20524
11. Farazi, S., Rafiei, D.: Top-k frequent term queries on streaming data. In: ICDE, pp. 1582–1585. IEEE (2019)

12. Felipe, I.D., Hristidis, V., Rishe, N.: Keyword search on spatial databases. In: Alonso, G., Blakeley, J.A. , Chen, A.L.P. (eds.) Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7–12, 2008, Cancún, Mexico, pp. 656–665. IEEE Computer Society (2008). https://doi.org/10.1109/ICDE.2008.4497474
13. Foursquare, https://foursquare.com/
14. Gao, Y., Qin, X., Zheng, B., Chen, G.: Efficient reverse top-k Boolean spatial keyword queries on road networks. IEEE Trans. Knowl. Data Eng. **27**(5), 1205–1218 (2015). https://doi.org/10.1109/TKDE.2014.2365820
15. Hadjieleftheriou, M., Kollios, G., Gunopulos, D., Tsotras, V.J.: On-line discovery of dense areas in spatio-temporal databases. In: Hadzilacos, T., Manolopoulos, Y., Roddick, J.F., Theodoridis, Y. (eds.) Advances in Spatial and Temporal Databases, 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 24–27, 2003, Proceedings, Lecture Notes in Computer Science, vol. 2750, pp. 306–324. Springer (2003). https://doi.org/10.1007/978-3-540-45072-6_18
16. Hoare, C.A.R.: Algorithm 65: find. Commun. ACM **4**(7), 321–322 (1961). https://doi.org/10.1145/366622.366647
17. Instragam, https://www.instagram.com/
18. Izbicki, M., Papalexakis, V., Tsotras, V.J.: Geolocating tweets in any language at any location. In: Zhu, W., Tao, D., Cheng, X., Cui, P., Rundensteiner, E.A., Carmel, D., He, Q., Yu J.X. (eds.) Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3–7, 2019, pp. 89–98. ACM (2019). https://doi.org/10.1145/3357384.3357926
19. Jeung, H., Yiu, M.L., Zhou, X., Jensen, C.S., Shen, H.T.: Discovery of convoys in trajectory databases. Proc. VLDB Endow. **1**(1), 1068–1080 (2008). https://doi.org/10.14778/1453856.1453971
20. Jonathan, C., Magdy, A., Mokbel, M.F., Jonathan, A.: GARNET: A holistic system approach for trending queries in microblogs, pp. 1251–1262 (2016). https://doi.org/10.1109/ICDE.2016.7498329
21. Korn, F., Muthukrishnan, S.: Influence sets based on reverse nearest neighbor queries. SIGMOD Rec. **29**(2), 201–212 (2000). https://doi.org/10.1145/335191.335415
22. Lappas, T., Arai, B., Platakis, M., Kotsakos, D., Gunopulos, D.: On burstiness-aware search for document sequences. In: IV, J.F.E., Fogelman-Soulié, F., Flach, P.A., Zaki, M.J. (eds.) Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28–July 1, 2009, pp. 477–486. ACM (2009). https://doi.org/10.1145/1557019.1557075
23. Lappas, T., Vieira, M.R., Gunopulos, D., Tsotras, V.J.: On the spatiotemporal burstiness of terms. Proc. VLDB Endow. **5**(9), 836–847 (2012)
24. Lu, J., Lu, Y., Cong, G.: Reverse spatial and textual k nearest neighbor search. In: Sellis, T.K., Miller, R.J., Kementsietsidis, A., Velegrakis, Y. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12–16, 2011, pp. 349–360. ACM (2011). https://doi.org/10.1145/1989323.1989361
25. Luo, C., Li, J., Li, G., Wei, W., Li, Y., Li, J.: Efficient reverse spatial and textual k nearest neighbor queries on road networks. Knowl. Based Syst. **93**, 121–134 (2016). https://doi.org/10.1016/j.knosys.2015.11.009
26. Ma, C., Lu, H., Shou, L., Chen, G.: KSQ: top-(k) similarity query on uncertain trajectories. IEEE Trans. Knowl. Data Eng. **25**(9), 2049–2062 (2013). https://doi.org/10.1109/TKDE.2012.152
27. Magdy, A., Aly, A.M., Mokbel, M.F., Elnikety, S., He, Y., Nath, S., Aref, W.G.: Geotrend: spatial trending queries on real-time microblogs, pp. 7:1–7:10 (2016). https://doi.org/10.1145/2996913.2996986
28. Mathioudakis, M., Bansal, N., Koudas, N.: Identifying, attributing and describing spatial bursts. Proc. VLDB Endow. **3**(1), 1091–1102 (2010). https://doi.org/10.14778/1920841.1920978
29. Ni, J., Ravishankar, C.V.: Pointwise-dense region queries in spatio-temporal databases. In: Chirkova, R., Dogac, A., Özsu, M.T., Sellis, T.K. (eds.) Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15–20, 2007, pp. 1066–1075. IEEE Computer Society (2007). https://doi.org/10.1109/ICDE.2007.368965
30. Nikitopoulos, P., Sfyris, G.A., Vlachou, A., Doulkeridis, C., Telelis, O.: Parallel and distributed processing of reverse top-k queries. In: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8–11, 2019, pp. 1586–1589. IEEE (2019). https://doi.org/10.1109/ICDE.2019.00148
31. Park, J.H., Chung, C.W., Kang, U.: Reverse nearest neighbor search with a non-spatial aspect. Inf. Syst. **54**(C), 92–112 (2015). https://doi.org/10.1016/j.is.2015.06.010
32. Qiao, B., Hu, B., Zhu, J., Wu, G., Giraud-Carrier, C.G., Wang, G.: A top-k spatial join querying processing algorithm based on spark. Inf. Syst. **87** (2020). https://doi.org/10.1016/j.is.2019.101419
33. Rocha-Junior, J.B., Vlachou, A., Doulkeridis, C., Nørvåg, K.: Efficient processing of top-k spatial preference queries. Proc. VLDB Endow. **4**(2), 93–104 (2010). https://doi.org/10.14778/1921071.1921076
34. Shang, S., Chen, L., Jensen, C.S., Wen, J., Kalnis, P.: Searching trajectories by regions of interest. IEEE Trans. Knowl. Data Eng. **29**(7), 1549–1562 (2017). https://doi.org/10.1109/TKDE.2017.2685504
35. Skovsgaard, A., Sidlauskas, D., Jensen, C.S.: Scalable top-k spatio-temporal term querying. In: Cruz, I.F. , Ferrari, E., Tao, Y., Bertino, E., Trajcevski, G. (eds.) IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31–April 4, 2014, pp. 148–159. IEEE Computer Society (2014). https://doi.org/10.1109/ICDE.2014.6816647
36. Stieglitz, S., Mirbabaie, M., Ross, B., Neuberger, C.: Social media analytics: challenges in topic discovery, data collection, and data preparation. Int. J. Inf. Manag. **39**, 156–168 (2018). https://doi.org/10.1016/j.ijinfomgt.2017.12.002
37. Tao, Y., Papadias, D., Lian, X.: Reverse knn search in arbitrary dimensionality. In: VLDB (2004)
38. tweetreach, https://tweetreach.com/
39. Twitter, http://twitter.com/
40. Twitter api, https://developer.twitter.com/
41. Uddin, M.R., Ravishankar, C.V., Tsotras, V.J.: Finding regions of interest from trajectory data. In: Zaslavsky, A.B., Chrysanthis, P.K., Lee, D.L., Chakraborty, D., Kalogeraki, V., Mokbel, M.F., Chow, C. (eds.) 12th IEEE International Conference on Mobile Data Management, MDM 2011, Luleå, Sweden, June 6–9, 2011, Volume 1, pp. 39–48. IEEE Computer Society (2011). https://doi.org/10.1109/MDM.2011.12
42. Uddin, M.R., Ravishankar, C.V., Tsotras, V.J.: Online identification of dwell regions for moving objects. In: Aberer, K., Joshi, A., Mukherjea, S., Chakraborty, D., Lu, H., Venkatasubramanian, N., Kanhere, S.S. (eds.) 13th IEEE International Conference on Mobile Data Management, MDM, pp. 248–257 (2012)
43. Uddin, R., Rice, M.N., Ravishankar, C.V., Tsotras, V.J.: Assembly queries: Planning and discovering assemblies of moving objects using partial information. In: Hoel, E.G., Newsam, S.D., Ravada, S., Tamassia, R., Trajcevski, G. (eds.) Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems,, pp. 24:1–24:10. ACM (2017)
44. Vieira, M.R., Bakalov, P., Tsotras, V.J.: On-line discovery of flock patterns in spatio-temporal data. In: Agrawal, D., Aref, W.G., Lu, C., Mokbel, M.F., Scheuermann, P., Shahabi, C., Wolfson, O. (eds.) 17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November

4–6, 2009, Seattle, Washington, USA, Proceedings, pp. 286–295. ACM (2009). https://doi.org/10.1145/1653771.1653812
45. Vlachou, A., Doulkeridis, C., Kotidis, Y., Nørvåg, K.: Reverse top-k queries. In: Li, F., Moro, M.M., Ghandeharizadeh, S., Haritsa, J.R., Weikum, G., Carey, M.J., Casati, F., Chang, E.Y., Manolescu, I., Mehrotra, S., Dayal, U., Tsotras, V.J. (eds.) Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1–6, 2010, Long Beach, California, USA, pp. 365–376. IEEE Computer Society (2010). https://doi.org/10.1109/ICDE.2010.5447890
46. Vlachou, A., Doulkeridis, C., Kotidis, Y., Nørvåg, K.: Monochromatic and bichromatic reverse top-k queries. IEEE Trans. Knowl. Data Eng. (2011)
47. Vlachou, A., Doulkeridis, C., Nørvåg, K.: Monitoring reverse top-k queries over mobile devices. In: Kollios, G., Tao, Y. (eds.) Proceedings of the Tenth ACM International Workshop on Data Engineering for Wireless and Mobile Access, MobiDE 2011, Athens, Greece, June 12, 2011, pp. 17–24. ACM (2011). https://doi.org/10.1145/1999309.1999313
48. Vlachou, A., Doulkeridis, C., Nørvåg, K., Kotidis, Y.: Identifying the most influential data objects with reverse top-k queries. Proc. VLDB Endow. **3**(1), 364–372 (2010). https://doi.org/10.14778/1920841.1920890
49. Vlachou, A., Doulkeridis, C., Nørvåg, K., Kotidis, Y.: Branch-and-bound algorithm for reverse top-k queries. In: Ross, K.A., Srivastava, D., Papadias, D. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22–27, 2013, pp. 481–492. ACM (2013). https://doi.org/10.1145/2463676.2465278
50. Wang, S., Bao, Z., Culpepper, J.S., Sellis, T., Sanderson, M., Qin, X.: Answering top-k exemplar trajectory queries. In: 33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19–22, 2017, pp. 597–608. IEEE Computer Society (2017). https://doi.org/10.1109/ICDE.2017.114
51. Wolframalpha, https://www.wolframalpha.com/
52. Yang, S., Cheema, M.A., Lin, X., Zhang, Y., Zhang, W.: Reverse k nearest neighbors queries and spatial reverse top-k queries. VLDB J. **26**(2), 151-176(2017). https://doi.org/10.1007/s00778-016-0445-2
53. Yiu, M.L., Dai, X., Mamoulis, N., Vaitis, M.: Top-k spatial preference queries. In: Chirkova, R., Dogac, A., Özsu, M.T., Sellis, T.K. (eds.) Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15–20, 2007, pp. 1076–1085. IEEE Computer Society (2007). https://doi.org/10.1109/ICDE.2007.368966
54. Yiu, M.L., Lu, H., Mamoulis, N., Vaitis, M.: Ranking spatial data by quality preferences. pp. 433–446 (2011). https://doi.org/10.1109/TKDE.2010.119
55. Younis, E.M.: Sentiment analysis and text mining for social media microblogs using open source tools: an empirical study. Int. J. Comput. Appl. **112**(5) (2015)
56. Zhao, J., Gao, Y., Chen, G., Jensen, C.S., Chen, R., Cai, D.: Reverse top-k geo-social keyword queries in road networks. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pp. 387–398 (2017). https://doi.org/10.1109/ICDE.2017.97
57. Zhou, X., Tao, X., Yong, J., Yang, Z.: Sentiment analysis on tweets for social events. In: Shen, W., Li, W., Barthès, J.A., Luo, J., Zhu, H., Yong, J., Li, X. (eds.) Proceedings of the 2013 IEEE 17th International Conference on Computer Supported Cooperative Work in Design (CSCWD), Whistler, BC, Canada, June 27–29, 2013, pp. 557–562. IEEE (2013). https://doi.org/10.1109/CSCWD.2013.6581022
58. Zhu, M., Papadias, D., Zhang, J., Lee, D.L.: Top-k spatial joins. IEEE Trans. Knowl. Data Eng. **17**(4), 567–579 (2005). https://doi.org/10.1109/TKDE.2005.65