Monitoring MaxRS in Spatial Data Streams

Daichi Amagata
Department of Multimedia Engineering Graduate School of Information Science and Technology
Osaka University
amagata.daichi@ist.osaka-u.ac.jp

Takahiro Hara
Department of Multimedia Engineering Graduate School of Information Science and Technology
Osaka University
hara@ist.osaka-u.ac.jp

ABSTRACT

Due to the increase of GPS enabled devices and a lot of location-based services, spatial objects are continuously generated. This paper addresses a problem of monitoring MaxRS (Maximizing Range Sum) in spatial data streams. Given a set of weighted spatial (2-dimensional) objects, this problem is to monitor a location of a given user-specified sized rectangle where the sum of the weights of the objects covered by the rectangle is maximized. Many real life applications obtain a benefit from monitoring MaxRS, e.g., traffic analysis and event detection in urban sensing, but this problem has not yet been addressed so far. Although some algorithms for static objects have been proposed, executing such an algorithm whenever new objects are generated is computationally expensive. These motivate us to develop an efficient algorithm that can monitor MaxRS efficiently. In this paper, we first design a basic algorithm that is based on an index framework and incrementally updates the result. We then enhance the algorithm and show that the enhanced algorithm can deal with error-guaranteed approximation and monitoring top-k MaxRS. Our experimental results confirm the efficiency of our approach.

1. INTRODUCTION

Due to the increase of GPS enabled devices such as smartphones and tablet machines, a lot of location-based services are used in many real life applications. From this fact, spatio-temporal databases have been receiving significant attention recently. Supporting spatial queries is therefore becoming more important, and many studies developed techniques for efficient spatial query processing [13, 15]. These studies can be classified into spatial objects retrieval problems [3, 24] and location finding problems [10, 32, 33]. For example, kNN query processing, which retrieves the k nearest neighbor objects w.r.t. a given query point, is the representative of the spatial objects retrieval problems. In the location finding problems, on the other hand, some queries have been proposed, for example, optimal location queries [10, 33], bichromatic reverse nearest neighbor queries [14, 35], and MaxRS (Maximizing Range Sum) queries [6, 8, 9, 25]. This paper focuses on a kind of MaxRS problem.

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

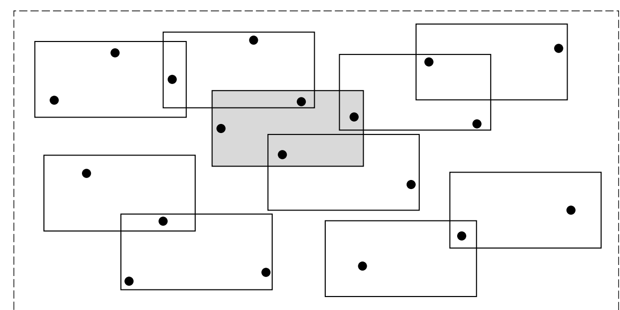


Figure 1: An example of a MaxRS query

Motivation. Given a set of weighted spatial (2-dimensional) objects and a user-specified sized rectangle, a MaxRS query finds a location of the rectangle where the sum of the weights of the objects covered by the rectangle is maximized.

EXAMPLE 1.1. *Figure 1 shows an example of a MaxRS query. In Figure 1, the rectangle with dashed line, the black points, and the rectangles with solid line show a general monitoring space, spatial objects with weight 1, and user-specified sized rectangles, respectively. The MaxRS query identifies the location of the shaded rectangle as one of the optimal results because it covers the largest number of objects, i.e., the sum of the weights of objects covered by the rectangle is the maximum.*

This query is useful because it can automatically find an important place without specifying any query points [8, 9]. In this paper, we address a novel problem of monitoring MaxRS in spatial data streams. In other words, we address a problem of continuous MaxRS query processing over sliding-window. Since a large amount of spatio-temporal objects are *continuously generated* [2, 5], e.g., in the context of location-based service usages, one-time finding an important location does not make sense but continuously monitoring an important location is required. A continuous MaxRS query can achieve this and has the following real life applications.

EXAMPLE 1.2. *Consider urban sensing. A system, e.g., base-station, continuously collects spatio-temporal objects, e.g., generated by devices with GPS, in an urban city. Such objects can be represented as $< x, y, w >$ where x, y, and w are latitude, longitude, and weight, respectively. If w is communication traffic, a continuous MaxRS query can monitor an area where traffic is concentrated. In this case, the system can notify the users holding mobile devices in the area of warning about communication delay.*

Example 1.2 shows that continuous MaxRS queries can help to analyze communication errors and also support decision making, e.g., where to place Wi-Fi access points. We next consider location-

based games in which continuous MaxRS queries are useful for decision making.

EXAMPLE 1.3. *In BotFighters, players try not to be attacked by other players [4], and in Ingress, players try to occupy places. Let w in Example 1.2 represent the strength or level of a given player, and players keep checking the area monitored by a continuous MaxRS query. This can detect an event, e.g., a competition by many and/or high-level players, and support to plan a strategy.*

Other examples include mobile sensor networks [27] that a base-station continuously collects sensor readings. By employing continuous MaxRS queries, the base-station may be able to detect sensor failures and over-concentration of mobile sensor nodes.

Technical challenge and overview. Some techniques for exact MaxRS query processing on static objects have been developed in the past. [12, 18] proposed in-memory algorithms while [8, 9] proposed an external-memory algorithm. The computation (I/O) complexity of the in-memory (external-memory) algorithm is shown to be optimal. These algorithms however focus on *one-time* computation, so they are not efficient for continuous MaxRS queries. This is because computing the result from scratch whenever new objects are generated is obviously computationally expensive, meaning that an approach which can incrementally update the result is required.

In this paper, we first propose a basic algorithm that exploits a graph in grid index, namely G2. G2 integrates graph and grid structures, thus its storage cost is $\mathcal{O}(|V|+|E|)$, where V corresponds to a set of objects on a given sliding-window and E is a set of edges. (The details are described in Section 4). One of the properties of G2 is that no overhead incurs when objects expire. We then enhance both the basic algorithm and G2, and propose aggregate G2, aG2, and a branch-and-bound algorithm that exploits aG2. This algorithm eliminates unnecessary update computation as much as possible and accelerates the query processing efficiency.

Interestingly, the branch-and-bound algorithm can deal with error-guaranteed approximation. Let w^* and ϵ respectively be the maximum range sum and a user-tolerance error rate. Also, let w be the weight of the area monitored by the algorithm, and we can guarantee that $w \geq (1-\epsilon)w^*$. We show that the relationship between query processing efficiency and ϵ is trade-off but the practical error rate is much less than ϵ. In addition to the approximation, we consider a problem of monitoring top-k MaxRS. For example, Examples 1.2 and 1.3 may require not only a single area but k (e.g., 5) areas with the largest range sum. This requirement is satisfied by a simple modification of the branch-and-bound algorithm.

Contributions and organization. We summarize our contributions as follows.

- We address a novel problem of continuous MaxRS query processing in spatial data streams (Section 2). To the best of our knowledge, we are the first to investigate this problem.
- We design a basic algorithm for a continuous MaxRS query (Section 4). This algorithm incrementally updates the result by exploiting an efficient index framework.
- We enhance the basic algorithm and propose a more efficient index and branch-and-bound algorithm (Section 5). This algorithm prunes unnecessary computation and accelerates the computation efficiency.
- We show that the branch-and-bound algorithm can deal with error-guaranteed approximation and efficient continuous top-k MaxRS query processing (Section 6).
- We conduct extensive experiments using both synthetic and real datasets that confirm the efficiency of our approach (Section 7).

In addition to the above contents, we review some related literatures in Section 3, and Section 8 concludes this paper.

2. PRELIMINARY

We are given a set of spatial stream objects in a general monitoring space, as shown in Figure 1. A spatial object o_i is represented by $o_i = < x, y, w >$ where i is its identifier, $< x, y >$ shows the location that o_i is generated, and w is a non-negative value (weight), i.e., $o_i.w \in \mathbb{R}^+$. We assume a spatial stream environment, thus consider a sliding-window model [1] because it is usual that many applications are interested only in recent objects. Count- and time-based sliding-window models are widely accepted. The count-based sliding-window considers the most recent n objects, and in this model, m new objects generations lead to the expirations of the m oldest objects. The time-based sliding-window, on the other hand, considers the objects generated within the last T time-units. Selection of a suitable sliding-window model depends on applications, and our algorithms can deal with both the models. So, without loss of generality, we assume the count-based sliding-window in this paper. Let O be the set of the most recent n objects, and O residents in-memory because real-time continuous query processing generally requires main memory computation [4, 17].

Let P be an infinite set of points in the general monitoring space. Given a user-specified sized rectangle r, the weight of a point $p \in P$, $p.w$, is defined by

$$p.w = \sum o_i.w,$$

where $o_i \in O$ is covered by r centered at p. We are now ready to formally define the monitoring MaxRS problem.

DEFINITION 1 (MONITORING MAXRS PROBLEM). *Given a set of objects on a sliding-window O, an infinite set of points in the general monitoring space, and a user-specified sized rectangle r, the goal of the monitoring MaxRS problem is to continuously monitor a location $p^* \in P$ that satisfies.*

$$p^* = \underset{p \in P}{\operatorname{argmax}}\ p.w.$$

It is infeasible to find and monitor p^* from such infinite points, but as the literatures [8, 9, 25] introduce, MaxRS problems can be solved by transformation to an alternative problem. Given a user-specified sized rectangle r, let r_i be the weighted rectangle of the same size of r centered at $< o_i.x, o_i.y >$ ($o_i \in O$ and $r_i.w = o_i.w$). Consider $o_i, o_j \in O$, and if r_i overlaps with r_j, it is not difficult to see that the weight of the overlapped space s is $s.w = r_i.w + r_j.w$. We formally define this *space weight.*

DEFINITION 2 (SPACE WEIGHT [8]). *Given O, we have a set of rectangles each of which is centered at the location of the corresponding object. The weight of a space s is the sum of the weights of the rectangles covering s.*

Then the alternative problem is to find s with the maximum weight denoted by s^*. Interestingly but not surprisingly, p^* exists in s^*, that is, any points in s^* can be p^*.

EXAMPLE 2.1. *Figure 2 shows the alternative problem of Figure 1. The center of each rectangle is at the corresponding object. Note that the size of each rectangle is the user-specified size. Recall that the weight of each object is 1, and the shaded overlapped space has*

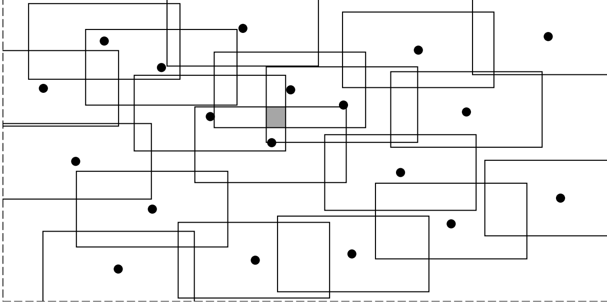


Figure 2: An example of the alternative problem of Figure 1

the maximum weight. We know that the weights of other overlapped spaces are less than 4. The center of the shaded rectangle in Figure 1 exists in the shaded space.

Therefore, monitoring s^* is equivalent to monitoring p^*. We here define continuous MaxRS queries that solve the monitoring MaxRS problem.

DEFINITION 3 (CONTINUOUS MAXRS QUERY). *Given a set of objects on a sliding-window O and a user-specified sized rectangle r, each weighted object is converted to a weighted rectangle centered at the location of the object. A continuous MaxRS query monitors s^* that satisfies*

$$s^* = \underset{s \in S}{\operatorname{argmax}}\ s.w, \quad (1)$$

where S is the set of overlapped spaces.

We also consider a problem of monitoring top-k MaxRS to support the applications that require not only a single space but also k spaces with the maximum weight. The definition of monitoring top-k MaxRS problem is given by extending Definition 1, thus we formally define continuous top-k MaxRS queries below.

DEFINITION 4 (CONTINUOUS TOP-K MAXRS QUERY). *Given a set of objects on a sliding-window O and a user-specified sized rectangle r, each weighted object is converted to a weighted rectangle centered at the location of the object. A continuous top-k MaxRS query monitors the set S^* that satisfies $|S^*| = k$ and $\forall s \in S^*$ and $\exists s' \in S \backslash S^*$, $s.w \geq s'.w$ where S is the set of overlapped spaces, and ties are arbitrarily broken.*

Since applications that execute continuous queries basically require real-time monitoring [3, 4, 17], algorithms that process such queries have to update query results efficiently. Our objective of this paper is therefore to minimize computation time to update s^*, which is incurred by generations and expirations of objects. Table 1 summarizes the symbols frequently used in this paper.

3. RELATED WORK

As introduced in Section 1, many spatial queries have been developed. This section reviews related works of the monitoring MaxRS problem. In particular, we introduce spatial preference queries and facility location queries in which MaxRS queries are categorized. We then review the existing studies of MaxRS query processing.

Before introducing the above queries, we should make it clear that MaxRS queries are different from range aggregation queries. The goal of range aggregation queries is to return the aggregate result (e.g., values and points) from (i) the set of points in a given rectangle with *fixed location* [19, 21, 23] or (ii) the set of values in a given interval [26]. On the other hand, the goal of MaxRS queries is to *find a location* from an infinite set of points. In the same sense, continuous spatial queries [15, 16, 17] that monitor not locations but objects are different from our problem.

Table 1: Overview of symbols

Symbol	Description
o_i	The weighted spatial object with identifier i
m	The number of objects generated at the same time
s^*	The space with the maximum weight
r_i	The weighted rectangle centered at the location of o_i
$r_i.E$	The set of edges held by r_i
$N(r_i)$	The set of neighboring vertices (rectangles) of r_i
s_i	The space with the maximum weight covered by r_i
$c_{i,j}$	The cell with identifier (i, j)
$G_{i,j}$	The graph maintained by $c_{i,j}$
$V_{i,j}$	The set of vertices (rectangles) of $G_{i,j}$
$c_{i,j}.w$	The upper-bound weight of $c_{i,j}$

Spatial preference queries. Spatial preference query processing problem is one of location selection problems. Given a 2-dimensional point p and a distance constraint d, a top-k spatial preference query [22, 29] determines the score of p by the sum of the weights of the feature objects existing within d from p. [28] studies a problem of finding top-t most influential sites. The influential score of a given site is defined as the sum of weights of its reverse nearest neighbor objects. In the above queries, the scores of the target points are defined by a kind of range sum function. However, the locations of the target points are already known, which is different from our problem.

Facility location queries. The objective of this problem is to find an optimal location w.r.t. a given condition. It is often the case that a set of customers (or clients) with weights and a set of facilities are given [11]. Then a facility location query retrieves a facility that maximizes the total weight of its reverse (k) nearest neighbor customers [35]. [10] proposed optimal location queries, and the extension version of the optimal location queries, min-dist optimal location queries, has been studied in [20, 32]. Such optimal location queries have also been studied in road networks [7].

The monitoring MaxRS problem is also one of the facility location problems. We are interested in continuously monitoring a location that maximizes the total weight of objects covered by a user-specified sized rectangle. As introduced in Section 1, this is useful in monitoring applications in spatial data streams. The main deference between MaxRS and the above queries is their categories: *monotonic* (e.g., MaxRS queries) and *bichromatic* (e.g., optimal location queries).

MaxRS queries. To the best of our knowledge, the existing works of MaxRS problems and the variants are [6, 8, 9, 12, 18, 25]. An external-memory algorithm for exact MaxRS queries has been proposed in [8, 9], while [25] proposed a randomized sampling algorithm that bounds the error with high probability. That is, given a tolerance error ϵ, the approximate algorithm returns a space with the weight w that satisfies $w \geq (1-\epsilon)w^*$ with probability $1 - \frac{1}{n}$. (w^* is the weight of the optimal result and n is the number of objects.) Rotating MaxRS queries have been proposed in [6]. This literature assumes that a given rectangle is rotatable, but we do not assume this case and assume the usual case, as well as [8, 9, 12, 18, 25].

We now focus on the in-memory algorithms [12, 18] because we also consider an in-memory algorithm for continuous MaxRS queries. It is notable that the algorithm in [18] solves a max-

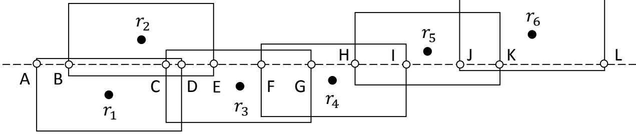


Figure 3: An example of processing a plane-sweep algorithm

enclosing rectangle problem, to which the MaxRS problem can be converted, based on well-known *plane-sweep* strategy, and actually the external-memory and the approximate algorithms [8, 25] also employ this strategy. (The algorithm proposed in [12] also employs the same strategy and the computation cost is the same as [18].) The plane-sweep algorithm [18] employs a horizontal line, and given a set of rectangles, this line is swept from bottom to top of the rectangles while counting the weights of intersecting intervals on the sweeping line. Figure 3 shows an instance when sweeping the dashed line. Let the weight of each rectangle in Figure 3 be 1, and we can see that the weights of intervals AB, BC, and CD are 1, 2, and 3, respectively. In the plane-sweep algorithm, when a sweeping line reaches the bottom edge of a rectangle, a newly generated interval is inserted into a binary-tree, while when reaching the top of the rectangle, the expired interval is deleted from the binary-tree. During these procedures, the counts of intervals are also updated. This algorithm returns the interval with the maximum weight, and the complexity of this algorithm is $\mathcal{O}(n \log n)$ [18], where n is the number of objects (rectangles). This is because the algorithm executes $2n$ binary-tree updates (n insertions and n deletions) that take $\mathcal{O}(\log n)$ time. (This algorithm also can return a set of k intervals with the maximum weight without sacrificing its computation efficiency.)

The above algorithm is shown to be an optimal solution for the MaxRS problem on *static objects*. However, it is inefficient to compute s^* from scratch by this algorithm even in the case where the number of newly generated objects is not large, which is shown by our experimental results. From the next section, we describe our solutions that efficiently update and monitor the result over stream data.

4. BASIC SOLUTION

The generations and the expirations of objects lead to additions and eliminations of overlapped spaces. This suggests that S in Equation (1) is dynamic, in other words, the ranking of range sum is dynamic, thereby efficient s^* monitoring is not trivial. In spite of this nature, our index framework enables to design a simple but efficient algorithm.

4.1 G2: Graph in Grid Index

We first present our index framework Graph in Grid index (or G2). Recall that our objective is to achieve real-time monitoring of MaxRS in a stream environment, meaning that unnecessary computations have to be avoided. A dynamic graph, where a vertex is a rectangle and an edge shows the overlap between given two vertices, realizes this. If the weight of each vertex (rectangle) is 1, we can see that s^* is a part of the vertex with the largest number of edges. To monitor s^* with using the dynamic graph, the basic operation that we have to do is just to check the updated parts of the dynamic graph. This is because only the updated parts may affect s^*. Motivated by this observation, we develop G2.

As described, the first idea is to maintain n rectangles on a sliding-window by a graph. Vertices of the graph are the rectangles, and if two rectangles overlap each other, there is an edge between them.

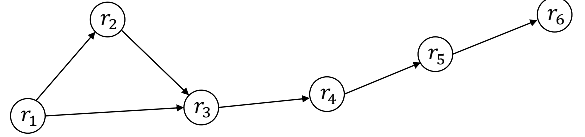


Figure 4: The graph constructed from the rectangles in Figure 3

Table 2: Edge and neighbor sets of the vertices of the graph in Figure 4

Vertex r_i	Edge $r_i.E$	Neighbor set $N(r_i)$
r_1	(r_1, r_2), (r_1, r_3)	$\{r_2, r_3\}$
r_2	(r_2, r_3)	$\{r_3\}$
r_3	(r_3, r_4)	$\{r_4\}$
r_4	(r_4, r_5)	$\{r_5\}$
r_5	(r_5, r_6)	$\{r_6\}$
r_6	$\emptyset$	$\emptyset$

We define our graph and provide a concrete example below.

DEFINITION 5 (GRAPH G). *$G = (V, E)$ is a dynamic graph. V is a set of rectangles of a given size on a sliding-window, and E is a set of direct edges. Given two vertices r_i, $r_j \in V$ overlapping each other, where r_i was generated earlier than r_j, there is a direct edge between them, (r_i, r_j), and this edge is held by the vertex (rectangle) that was generated earlier than the other, i.e., r_i. (Ties are broken by identifiers.)*

EXAMPLE 4.1. *We construct a graph G from the set of rectangles in Figure 3, and Figure 4 shows G. Assume that the rectangles are generated in order of identifiers. Since r_2 overlaps with r_1, there is (r_1, r_2) that represents a direct edge from r_1 to r_2. r_1 maintains the edges (r_1, r_2) and (r_1, r_3).*

When context is clear, we use vertices and rectangles interchangeably since they are the same, as Definition 5 describes. We denote $r_i.E$ the set of edges held by the vertex $r_i \in V$. The set of neighbors of a vertex $r_i \in V$ is also denoted by $N(r_i) = \{\forall r_j \,|\, \exists (r_i, r_j) \in r_i.E\}$. Table 2 shows an example that uses Figure 4 in terms of $r.E$ and $N(r)$. Then we obtain the spaces on r_i covered by the rectangles in $N(r_i)$. r_i maintains s_i, the space with the maximum weight among the spaces. That is, s_i is definitely the subspace on r_i, which provides the following property.

PROPERTY 1. *Given $r_i, r_j \in V$, we have that $s_i \neq s_j$.*

The proof is straightforward since $\nexists r_i \in N(r_j)$ if $\exists r_j \in N(r_i)$. How to obtain s_i is explained later, but it is not difficult to see the following property.

PROPERTY 2. *Let S_G be the set of s_i maintained by r_i on a sliding-window, we have that*

$$s^* = \operatorname*{argmax}_{s_i \in S_G} s_i.w. \tag{2}$$

Furthermore, even if some vertices (rectangles) expire, other vertices need no maintenances, because the corresponding edges are held by older vertices.

PROPERTY 3. *Given $r_i \in V$, and when the older vertices than r_i expire, s_i does not change.*

We next have to consider the case where m new rectangles (objects) are generated. Due to the graph structure, we need to check whether the m rectangles overlap with the existing rectangles. It is obvious that simple computation takes $\mathcal{O}(mn)$ time. To alleviate this, we employ a grid structure like the one shown in Figure 5,

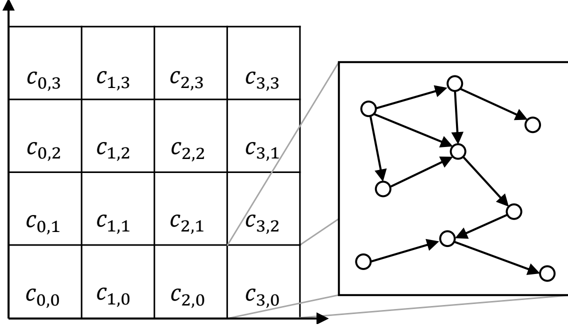


Figure 5: An example of G2

which is the second idea. When dataset updates frequently occur, grid structure is more suitable than complex structures like R-tree and Quad-tree [4]. Each cell in the gird is assigned an identifier and its size is fixed, as shown in Figure 5. Note that each cell $c_{i,j}$ maintains the graph constructed by the mapped rectangles, denoted by $G_{i,j} = (V_{i,j}, E_{i,j})$. Therefore, Definition 5 is rewritten as follows.

DEFINITION 6 (GRAPH $G_{i,j}$). *$G_{i,j} = (V_{i,j}, E_{i,j})$ is a dynamic graph maintained by $c_{i,j}$ of a grid. $V_{i,j}$ is a set of given sized rectangles that are mapped to $c_{i,j}$ and on a sliding-window, and $E_{i,j}$ is a set of direct edges. The condition of direct edges follows Definition 5.*

For example, Figure 5 illustrates an example of a G2 and $G_{3,0}$. When m new rectangles are generated, we map them to the cells with which they overlap. (So, a new rectangle may be mapped to multiple cells.) We then update the graphs in the cells where new objects are mapped.

Complexities. The time complexity of the rectangle mapping is $\mathcal{O}(m)$. Let c' and m' respectively be the number of cells where new objects are mapped and the average number of new objects mapped to the cells. Also, let n' be the average number of vertices in the cells where new objects are mapped, and the time complexity of the graph update is $\mathcal{O}(c'm'n')$. In practice, we have that $n' \ll n$. We next consider storage cost. Let V be the set of all vertices in the grid, then we know that $|V| = \sum |V_{i,j}|$. Similarly, $|E| = \sum |E_{i,j}|$, where E is the set of all edges in the grid. Recall that each vertex r_i maintains s_i and its storage cost is $\mathcal{O}(|V|)$. We can therefore conclude that the storage cost of G2 is $\mathcal{O}(|V| + |E|)$.

4.2 Monitoring Algorithm with G2

We design an online monitoring algorithm using G2, and Algorithm 1 illustrates the high level algorithm.

Algorithm description. Consider an instance when m new rectangles are generated and the m oldest rectangles expire. As described in Section 4.1, we first update G2 (lines 1–3). Lines 4–6 are designed based on the following idea. Given the set S of the spaces each of which (i.e., s_i) is maintained by $r_i \in V$, we know that s_i has to be correct due to Equation (2). In addition, if a new edge is inserted to $r_i.E$, s_i may change because $N(r_i)$ varies. We therefore need to compute s_i if $r_i.E$ is updated. The plane-sweep algorithm is an optimal solution to find the space with the maximum weight covered by the given rectangles [12]. Hence, we employ the plane-sweep algorithm to compute s_i (line 6). Note that the input of this plane-sweep algorithm is *only* $N(r_i) \cup \{r_i\}$. To summarize, for $\forall r_i \in V$, where $r_i.E$ has new edges and V is the vertex set maintained by a given cell c of G2, Algorithm 1 executes the plane-sweep algorithm locally, denoted by Local-Plane-Sweep($\cdot$) (lines 4–6). After that, we can correctly monitor s^*.

Algorithm 1: Monitoring algorithm using G2

```
1 Mapping(R) // R is the set of new rectangles
2 C' ← the set of the cells where new objects are mapped
3 G2-Update(C') // rectangle overlap computation
4 for ∀c ∈ C' do
5   for ∀r_i ∈ c.V where r has new edges do
6     s_i ← Local-Plane-Sweep(N(r_i) ∪ {r_i})
7 s* ← argmax_{s_i∈S} s_i.w
8 return s*
```

Recall that given a set of rectangles, the plane-sweep algorithm sweeps a horizontal line from the bottom to the top among the rectangles. To obtain s_i, however, we only need to sweep the horizontal line from the bottom to the top of r_i. Local-Plane-Sweep($\cdot$) in Algorithm 1 (line 6) is optimized to do so. Although the time complexity does not vary, the practical execution time is reduced.

The following simple example highlights the efficiency of our incremental approach.

EXAMPLE 4.2. *Assume that the graph in Figure 4 is the graph maintained by one of the cells in Figure 5. Assume further that $m = 1$ and r_6 is the new rectangle mapped to the cell. Algorithm 1 checks the vertices overlapping with r_6, and in this case, (r_5, r_6) is inserted to $r_5.E$. Algorithm 1 next executes Local-Plane-Sweep($\{r_5, r_6\}$) and obtains s_5. If $s_5.w > s^*.w$, s^* is replaced by s_5.*

Time complexity. As discussed before, lines 1–3 take $\mathcal{O}(c'm'n')$ time. Let v be the number of vertices to which new edges are inserted. Also, let e be the average number of edges of the above vertices, then lines 4–6 take $\mathcal{O}(ve \log e)$ time since Local-Plane-Sweep($\cdot$) for a vertex takes $\mathcal{O}(e \log e)$ time. Therefore, when m new objects are generated, Algorithm 1 takes $\mathcal{O}(c'm'n' + ve \log e)$ time.

5. ENHANCED SOLUTION

Algorithm 1 can identify *where to update* and compute s_i maintained by r_i efficiently since $N(r_i)$ is easily obtained by the graph representation. In fact, however, the most time consuming operation in Algorithm 1 is Local-Plane-Sweep($\cdot$) (line 6). Algorithm 1 executes Local-Plane-Sweep($\cdot$) whenever $r_i.E$ is updated, thus it is intuitively seen that the approach degrades the performance in the case where the number of $r.E$ updates is large. We observe the following usual cases that motivate to enhance G2.

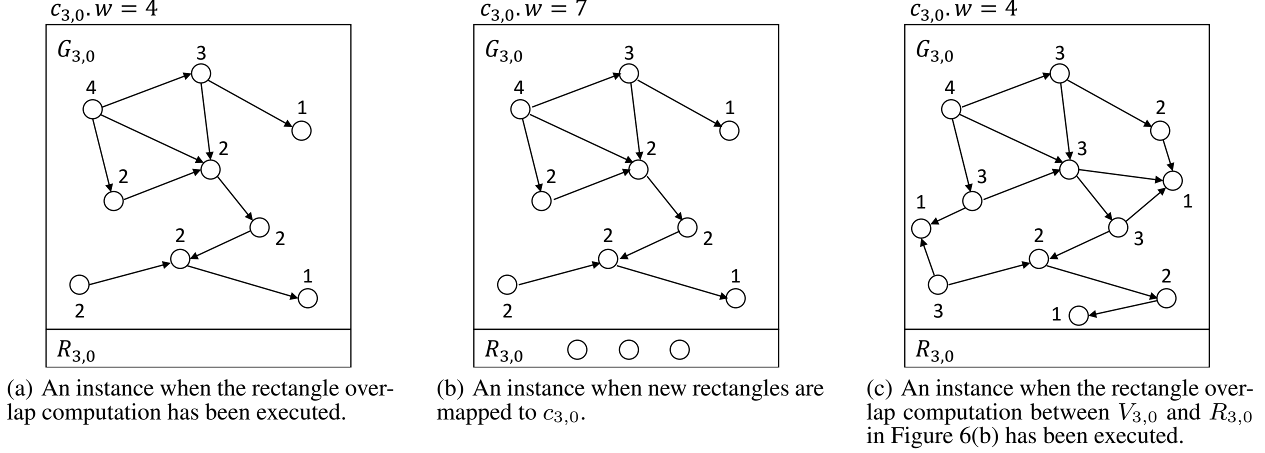
1. The $r_i.E$ update increases $s_i.w$ but it is less than $s^*.w$.

2. The $r_i.E$ update does not increase $s_i.w$.

To consider a more concrete situation, we give Example 5.1, and Table 3 shows the weights of each vertex r_i and s_i of the graph in Figure 4.

Table 3: Weights of vertex r_i and s_i of the graph in Figure 4

Vertex r_i	$r_i.w$	$s_i.w$
r_1	10	55
r_2	30	45
r_3	15	40
r_4	25	45
r_5	20	25
r_6	5	5



(a) An instance when the rectangle overlap computation has been executed.

(b) An instance when new rectangles are mapped to $c_{3,0}$.

(c) An instance when the rectangle overlap computation between $V_{3,0}$ and $R_{3,0}$ in Figure 6(b) has been executed.

Figure 6: An example of $G_{3,0}$ (followed by Figure 5) and $R_{3,0}$ of $c_{3,0}$ in an aG2 and its dynamic update where the weight of each vertex is 1.

EXAMPLE 5.1. *Assume the same situation as Example 4.2 and $s^* = s_1$. Before (r_5, r_6) is inserted to $r_5.E$, $s_5.w = 20$, and after the insertion, we obtain $s_5.w = 25$. However, it is obvious that the edge (r_5, r_6) insertion to $r_5.E$ does not affect s^*, which corresponds to the case 1. In addition, even if r_6 overlaps with r_2 and does not overlap with r_3, s_2 keeps the same and does not become s^*. This corresponds to the case 2.*

The observation in Example 5.1 suggests that we may not have to compute s_i even when $r_i.E$ is updated. Therefore, by enhancing G2, we aim at eliminating such unnecessary computation and improving query processing efficiency. The enhanced solution achieves this by an upper-bounding technique.

5.1 Aggregate G2

It has been shown in the past that data structures considering aggregate results work well for query processing which deal with aggregate functions such as `sum` and `count` [30]. The monitoring MaxRS problem considers `sum` function, then we know that G2 can be extended to deal with aggregate values like aR-tree [19].

We propose aG2 (aggregate G2), which is essentially G2. The main difference between G2 and aG2 is that aG2 employs *upper-bound weights*. Given a graph in an aG2, each vertex r_i of the graph maintains $s_i.\overline{w}$, which is the upper-bound weight of s_i. How to compute $s_i.\overline{w}$ depends on algorithms, so we briefly introduce our approach to compute $s_i.\overline{w}$ here (the detail is described in Section 5.2). Given the graph $G_{i,j} = (V_{i,j}, E_{i,j})$ maintained by a cell $c_{i,j}$ in an aG2, we have a vertex $r_{i'} \in V_{i,j}$ and $s_{i'}$. Consider rectangles $r_{j'}$ where $(r_{i'}, r_{j'})$ is newly inserted to $r_{i'}.E$, $s_{i'}.\overline{w}$ is computed by the following Equation.

$$s_{i'}.\overline{w} = s_{i'}.w + \sum r_{j'}.w \tag{3}$$

Not only the vertices but also the cells in aG2 maintain upper-bound weights. Before we introduce the upper-bound weight maintained by $c_{i,j}$, we have to note that in aG2, $c_{i,j}$ maintains

- $G_{i,j}$: the graph defined by Definition 6, and
- $R_{i,j}$: a set of rectangles that have been mapped to $c_{i,j}$ but have not yet been checked whether they overlap with vertices in $V_{i,j}$ or not (denoted by rectangle overlap computation).

The new rectangles mapped to $c_{i,j}$ are initially maintained in $R_{i,j}$. When we execute the rectangle overlap computation, the rectangles in $R_{i,j}$ are moved to $V_{i,j}$. Now we introduce how to compute $c_{i,j}.w$, i.e., the upper-bound weight maintained by $c_{i,j}$. Basically, $c_{i,j}.w$ is set as follows.

$$c_{i,j}.w = \max_{s_{i'} \in S_{i,j}} s_{i'}.\overline{w}, \tag{4}$$

where $S_{i,j}$ is the set of $s_{i'}$ maintained by $r_{i'} \in V_{i,j}$. Given a set of new rectangles r' that are mapped to $c_{i,j}$, $c_{i,j}.w$ is updated by the following equation.

$$c_{i,j}.w \leftarrow c_{i,j}.w + \sum r'.w \tag{5}$$

We give a simple example of how to dynamically update upper-bound weights below.

EXAMPLE 5.2. *Figure 6 shows an example of $G_{3,0}$ and $R_{3,0}$ where $c_{3,0}$ follows Figure 5. We first assume the situation of Figure 6(a), and note that the value shown next to each vertex is the upper-bound weight. Then we see that $c_{3,0}.w = 4$, which is the maximum value among the upper-bound weights of $V_{3,0}$. Next we assume that three new rectangles are mapped to $c_{3,0}$, which are maintained in $R_{3,0}$, as shown in Figure 6(b). From Equation (5), $c_{3,0}.w$ is updated to 7. We finally assume Figure 6(c), where the three rectangles in $R_{3,0}$ have been checked whether they overlap with the vertices (rectangles) in $V_{3,0}$ or not. Note that the upper-bound weight maintained by each vertex is updated and $c_{3,0}.w$ is also updated to 4 as the result of the rectangle overlap computation.*

From Equations (3)–(5), we have the following property.

PROPERTY 4. *$c_{i,j}.w \geq s_{i'}.\overline{w} \geq s_{i'}.w$ for $\forall r_{i'} \in V_{i,j}$.*

Because of generations and expirations of rectangles, those upper-bound weights may vary dynamically, but our branch-and-bound algorithm (introduced later) dynamically updates the upper-bound weights so that Property 4 is kept. At the same time, this property provides the correctness of our branch-and-bound algorithm.

As well as G2, we discuss about the storage cost of an aG2 below.

PROPERTY 5. *An aG2 has the same storage cost as a G2, i.e., $\mathcal{O}(|V| + |E|)$ where V and E are respectively the sets of all the vertices and the edges in the (a)G2.*

PROOF. To prove Property 5, we have to discuss about the storage costs of vertices, edges, and upper-bound weights. Let $n_{i,j}$ and $n'_{i,j}$ be the size of $V_{i,j}$ in G2 and aG2, respectively. Because $V_{i,j} \cap R_{i,j} = \emptyset$ in aG2, we have that $n_{i,j} = n'_{i,j} + |R_{i,j}|$, thereby $|V| = \sum(n'_{i,j} + |R_{i,j}|)$. The number of edges in aG2 may be less than that of G2 but it takes $\mathcal{O}(E)$ cost. Let V' be the set of $V_{i,j}$ of

aG2, and $|V| > |V'| = \sum n'_{i,j}$. The storage cost of upper-bound weights maintained by vertices is $\mathcal{O}(|V'|)$. Let C be the set of cells of an aG2, and the storage cost of upper-bound weights maintained by the cells is $\mathcal{O}(|C|)$. We know that $|C| \ll |V'|$ in practice. Therefore we can complete the proof. □

5.2 Branch-and-Bound Algorithm

As described in Section 5.1, each cell and vertex in an aG2 maintain the upper-bound weights. This enables us to process a continuous MaxRS query while pruning unnecessary computations. Specifically, we can obtain two pruning rules, which can reduce the number of executions of Local-Plane-Sweep($\cdot$). Assume an instance when m new rectangles are generated, the upper-bound weight of each cell is obtained by Equation (5). Given s^*, we first obtain the following pruning rule.

PRUNING RULE 1. *Given a cell $c_{i,j}$ in the aG2, and if $c_{i,j}.w < s^*.w$, all the vertices in $V_{i,j}$ do not have s^*. Therefore we do not need to compute the exact $s_{i'}$ of $r_{i'} \in V_{i,j}$.*

The above case efficiently prunes the computation of the exact $s_{i'}$. However, we are likely to hold the case where $c_{i,j}.w \geq s^*.w$, since $c_{i,j}.w$ is the maximum value among the set of the upper-bounds maintained by vertices in $G_{i,j}$ or more (See Equation(5)). In this case, we focus on each vertex in $V_{i,j}$, and then apply the next pruning rule.

PRUNING RULE 2. *Given a cell $c_{i,j}$ in the aG2, and we assume that $c_{i,j}.w \geq s^*.w$. Given a vertex $r_{i'} \in V_{i,j}$, if $s_{i'}.\overline{w} < s^*.w$, $r_{i'}$ does not have s^*, thus we do not need to compute the exact $s_{i'}$.*

From the above pruning rules, we can focus only on cells and vertices with non-zero probability to have s^*. Note that the above pruning rules assume that s^* is given, although s^* might expire. If s^* expires, we first obtain a temporal s^*, and the temporal s^* is retrieved from the cell c that satisfies

$$c = \operatorname*{argmax}_{c_{i,j} \in C} c_{i,j}.w, \qquad (6)$$

where C is the set of cells in the aG2. To keep the efficiency of the pruning rules, the weight maintained by the temporal s^* should be large as much as possible. It is intuitive that a cell with large upper-bound weight probably has the space with large weight, thereby we employ this heuristic. From the above discussion, we design a branch-and-bound algorithm, which efficiently updates s^* and is illustrated in Algorithm 2.

Algorithm description. Consider an instance when m new rectangles are generated and the m oldest rectangles expire. We first map the newly generated rectangles to the corresponding cells $c_{i,j}$ while updating the upper-bound weight $c_{i,j}.w$ and $R_{i,j}$ (lines 1–5). Next, we update (or find a temporal) s^* to enhance the pruning efficiency (line 6–10). Let c is the cell holding s^* (if s^* expires, we retrieve c that satisfies Equation (6)), we execute OverlapComputation(c) (line 9), which is illustrated in Algorithm 3. OverlapComputation(c) updates the graph in c and $c.w$. More specifically, we check whether rectangles in R of the cell c overlap with the rectangles (vertices) in V, and if overlap, new edges are inserted while updating the upper-bound weights maintained by the vertices and c (lines 3–8 in Algorithm 3). After that, ExactWeightComputation(s^*, c) is executed (line 10 in Algorithm 2), which is illustrated in Algorithm 4. In ExactWeightComputation(s^*, c), we apply Pruning rule 2 to each vertex r_i. If $s_i.\overline{w} > s^*.w$, we execute Local-Plane-Sweep($\cdot$) for r_i, and update s^* if necessary (lines 6–10 in Algorithm 4). Also, $c.w$ is kept so that it satisfies Equation (4). From the above procedures, we obtain the updated (or temporal) s^*, and then a branch-and-bound approach is employed (lines 11–15 in Algorithm 2) to guarantee the correct s^*.

Algorithm 2: Branch-and-bound algorithm using aG2

```
1  R_new ← the set of newly generated rectangles
2  for ∀r ∈ R_new do
3      if r is mapped to c_{i,j} then
4          c_{i,j}.w ← c_{i,j}.w + r.w
5          R_{i,j} ← R_{i,j} ∪ {r}
6  c ← {c_{i,j} | s* is in c_{i,j}}
7  if s* expired (c = ∅) then
8      c ← argmax_{c_{i,j} ∈ C} c_{i,j}.w // C is the set of cells in aG2
9  OverlapComputation(c)
10 s* ← ExactWeightComputation(s*, c)
11 for ∀c_{i,j} ∈ C\{c} do
12     if c_{i,j}.w > s*.w then
13         OverlapComputation(c_{i,j})
14     if c_{i,j}.w > s*.w then
15         s* ← ExactWeightComputation(s*, c_{i,j})
16 return s*
```

Algorithm 3: OverlapComputation($c_{i,j}$)

```
   Input: c_{i,j} // a cell in aG2
1  c_{i,j}.w ← 0
2  for ∀r' ∈ R_{i,j} do
3      for ∀r ∈ V_{i,j} do
4          if r' overlaps with r then
5              r.E ← r.E ∪ {(r, r')}
6              s.w̄ ← s.w̄ + r'.w
7          if c_{i,j}.w < s.w̄ then
8              c_{i,j}.w ← s.w̄
9      if c_{i,j}.w < r'.w then
10         c_{i,j}.w ← r'.w
11     s'.w̄ ← r'.w
12     V_{i,j} ← V_{i,j} ∪ {r'}
13     R_{i,j} ← R_{i,j}\{r'}
```

Given a cell $c_{i,j}$, we first apply Pruning rule 1, and if $c_{i,j}.w > s^*.w$, we update (i.e., decrease) $c_{i,j}.w$ by OverlapComputation($c_{i,j}$) (lines 12–13). Again we apply Pruning rule 1 to $c_{i,j}$, and if $c_{i,j}.w > s^*.w$ again, ExactWeightComputation($s^*, c_{i,j}$), which we explained above, is executed. In the case where we compute the exact $s_{i'}$ maintained by $r_{i'} \in V_{i,j}$, in ExactWeightComputation($s^*, c_{i,j}$), and $s_{i'}.w > s^*.w$, s^* is updated (line 10 in Algorithm 4). These operations are executed for $\forall c_{i,j} \in C \backslash \{c\}$, and after that, we can keep monitoring the correct s^*.

Time complexity. OverlapComputation($c_{i,j}$) takes $\mathcal{O}(|V_{i,j}||R_{i,j}|)$. Let C' be the set of the cells that OverlapComputation($\cdot$) is executed, and the amortized time to compute the rectangle overlapping is $\mathcal{O}(|C'||V_{i,j}||R_{i,j}|)$. Let v' be the number of the vertices that Local-Plane-Sweep($\cdot$) is executed. Also let e' be the average number of edges of the above vertices, and the total cost of ExactWeightComputation($\cdot,\cdot$) is $\mathcal{O}(v'e'\log e')$. When m new objects are generated, Algorithm 2 takes $\mathcal{O}(|C'||V_{i,j}||R_{i,j}|+v'e'\log e')$ (amortized) time. Recall the time complexity of Algorithm 1, and note that $|C'||V_{i,j}||R_{i,j}| \leq c'm'n'$ and $v'e'\log e' < ve\log e$.

Correctness. The correctness of Algorithm 2 is proven by Property 4. In Algorithms 2–4, we define the condition that we cannot prune OverlapComputation($\cdot$) and ExactWeightComputation($\cdot,\cdot$) as "$>$" instead of "$\geq$," e.g., line 12 of Algorithm 2. This is because

Algorithm 4: ExactWeightComputation($s^*, c_{i,j}$)

```
Input: s*, c_{i,j} // c_{i,j} is a cell in aG2
1  c_{i,j}.w ← 0
2  for ∀r_{i'} ∈ V_{i,j} do
3      ρ ← 0
4      if s* ≠ ∅ then
5          ρ ← s*.w
6      if s_{i'}.w̄ > ρ then
7          s_{i'} ← Local-Plane-Sweep(N(r_{i'} ∪ {r_{i'}}))
8          s_{i'}.w̄ ← s_{i'}.w
9          if s_{i'}.w > s*.w then
10             s* ← s_{i'}
11     if c_{i,j}.w < s_{i'}.w̄ then
12         c_{i,j}.w ← s_{i'}.w̄
13 return s*
```

Algorithm 5: UpperboundUpdate(r_i)

```
Input: r_i // a vertex of a given cell.
1  R(r_i) ← the set of vertices that are included in N(r_i) but have not
   been executed Plane-Sweep(·)
2  τ ← s_i.w
3  for ∀r ∈ R(r_i) do
4      if r overlaps with s_i then
5          τ ← τ + r.w
6          if τ > s*.w then
7              s_i.w̄ ← τ
8              break
9      else
10         ρ ← r.w + r_i.w
11         for r' ∈ N(r_i)\{r} do
12             if r overlaps with r' then
13                 ρ ← ρ + r'.w
14         if τ < ρ then
15             τ ← min(τ + r.w, ρ)
16             if τ > s*.w then
17                 s_i.w̄ ← τ
18                 break
```

we monitor one of the spaces with the maximum weight, thus if $s^*.w = c_{i,j}.w$ for example, we keep monitoring s^*, and this does not sacrifice the correctness. If applications require all spaces with the maximum weight like the AllMaxRS problem [9], we just need to define the condition as "$\geq$."

5.3 Discussion

To demonstrate the efficiency and practicality of Algorithm 2, we review the following conceivable approaches that can tight the upper-bound weights maintained by rectangles more than Algorithm 2.

1. An approach that all rectangles r_i maintain *all* the overlapped spaces on r_i.
2. An additional approach of Algorithm 2 that computes the maximum space weight among the common spaces on r_i and r_j when (r_j, r_i) is inserted to $r_i.E$.
3. An additional approach of Algorithm 2 that tries to decrease $s_i.\overline{w}$ when $s_i.\overline{w} > s^*.w$.

We show that these approaches do not guarantee the reduction of time complexity, rather, the worst-case time complexity becomes worse than Algorithm 2.

Approach 1. Given a rectangle r_i, we assume that S_i is the set of the overlapped spaces on r_i. Since r_i maintains all the overlapped spaces on r_i, we can compute the tightest $s_i.\overline{w}$. However, given m' new rectangles overlapping with r_i, we need at least $\mathcal{O}(m' \log |S_i|)$ time in practice[1] to compute the tightest $s_i.\overline{w}$. Because $|S_i| \geq |r_i.E|$, we may have that $\mathcal{O}(m' \log |S_i|)$ is larger than the time cost of the plane-sweep algorithm, i.e., $\mathcal{O}(|r_i.E| \log |r_i.E|)$. To bound the worst time complexity, we execute Local-Plane-Sweep($\cdot$), if $\mathcal{O}(m' \log |S_i|) > \mathcal{O}(|r_i.E| \log |r_i.E|)$ (if we can estimate this situation). Then the worst-case time complexity is the same as Algorithm 1, thus is worse than Algorithm 2. Note that this approach is impractical because we cannot bound the number of spaces maintained by rectangles.

Approach 2. Recall that an optimal way to compute the maximum space weight is the plane-sweep algorithm, thereby this approach is equivalent to the exact s_i computation. That is, this approach does not make sense.

Approach 3. Given a vertex r_i in an aG2, and when $s_i.\overline{w} > s^*.w$, this approach tries to decrease $s_i.\overline{w}$. This approach is illustrated by Algorithm 5, which may be executed after line 6 of Algorithm 4. Assume an instance when s_i is computed, and let $R(r_i)$ be the set of vertices that are included in $N(r_i)$ after the instance. In other words, s_i has been computed based on $N(r_i)\backslash R(r_i)$. In a nut shell, this approach computes $s_i.\overline{w}$ by checking whether a given $r \in R(r_i)$ overlaps with s_i (line 4) or the rectangle $r' \in N(r)\backslash\{r\}$ (line 12). Due to the latter case, this approach does not necessarily add $r.w$ to $s_i.\overline{w}$, which may result in less $s_i.\overline{w}$ than the original Algorithm 4. This approach however needs an extra $\mathcal{O}(|R(r_i)||N(r_i)|)$ time *for each* r_i where $s_i.\overline{w} > s^*.w$. Recall that Local-Plane-Sweep($N(r_i) \cup \{r_i\}$) takes $\mathcal{O}(|N(r_i)| \log |N(r_i)|)$ time, and we may have that $\mathcal{O}(|R(r_i)||N(r_i)|) > \mathcal{O}(|N(r_i)| \log |N(r_i)|)$. It would be better to execute Algorithm 5 only in the case where $|R(r_i)||N(r_i)| < 2|N(r_i)| \log |N(r_i)|$ (since the plane-sweep algorithm needs $2n \log n$ operations). We can therefore see that this approach does not work well if $|R(r_i)|$ is large. Also, this approach increases the worst-case time complexity, which means no guarantee to reduce the time complexity of Algorithm 2. Our experimental results also show that this approach does not guarantee the acceleration of query processing.

From the above discussion, we see that more storage and non-reasonable computational costs are required to tight the upper-bound weights. The upper-bounding cost of Algorithm 2 is reasonable and the worst-case time complexity is better than the above approaches.

6. APPLICATION TO RELATED PROBLEMS

In this section, we address two problems of approximate monitoring MaxRS and monitoring top-k MaxRS. We solve these problems efficiently by employing the branch-and-bound algorithm using aG2 with simple extensions.

6.1 Approximate Monitoring MaxRS

To improve query processing efficiency, some applications require not the exact but approximate results [25]. In this case, it is important to bound the error rate, thus, given a user-tolerance error ϵ ($0 \leq \epsilon < 1$), the objective of this problem is to continuously monitor a space s with the weight $s.w$ that satisfies

$$s.w \geq (1 - \epsilon)s^*.w.$$

[1]In the case where S_i is indexed by an R-tree or a Quad-tree.

Algorithm 2 can deal with this problem by respectively replacing Pruning rules 1 and 2 with Pruning rules 3 and 4, which are shown below.

PRUNING RULE 3. *Given a cell $c_{i,j}$ in the aG2 and the space s monitored by our approximate algorithm, if $(1-\epsilon)c_{i,j}.w < s.w$, we do not compute the exact $s_{i'}$ of $r_{i'} \in V_{i,j}$.*

PRUNING RULE 4. *Given a cell $c_{i,j}$ in the aG2 and the space s monitored by our approximate algorithm, if $(1-\epsilon)c_{i,j}.w \geq s.w$, we cannot prune OverlapComputation$(c_{i,j})$ by Pruning rule 3. Given a vertex $r_{i'} \in V_{i,j}$, if $(1-\epsilon)s_{i'}.\overline{w} < s.w$, we do not compute the exact $s_{i'}$.*

We demonstrate that our approximate algorithm (which is the approximate version of Algorithm 2) guarantees the error bound.

THEOREM 1. *Our approximate algorithm always guarantees that $s.w \geq (1-\epsilon)s^*.w$.*

To prove Theorem 2, we need to introduce the following lemmas.

LEMMA 1. *Assume that we are monitoring s that satisfies $s.w \geq (1-\epsilon)s^*.w$ before applying Pruning rule 3 to a cell $c_{i,j}$. It is guaranteed that Pruning rule 3 does not lose that $s.w \geq (1-\epsilon)s^*.w$.*

PROOF. If $c_{i,j}.w < s^*.w$ and $(1-\epsilon)c_{i,j}.w < s.w$, it is trivial that $(1-\epsilon)s^*.w \leq s.w$ due to the assumption. On the other hand, if $c_{i,j}.w \geq s^*.w$, we have that $(1-\epsilon)c_{i,j}.w \geq (1-\epsilon)s^*.w$. Therefore if $s.w > (1-\epsilon)c_{i,j}.w$, $s.w > (1-\epsilon)s^*.w$. □

LEMMA 2. *Assume that we are monitoring s that satisfies $s.w \geq (1-\epsilon)s^*.w$ before applying Pruning rule 4 to a vertex $r_{i'}$. It is guaranteed that Pruning rule 4 does not lose that $s.w \geq (1-\epsilon)s^*.w$.*

PROOF. Essentially the same as the proof of Lemma 1. □

Now, we are ready to prove Theorem 1.

PROOF. As long as we are monitoring s that satisfies $s.w \geq (1-\epsilon)s^*.w$, Pruning rules 3 and 4 do not lose the bound, which can be seen from Lemmas 1 and 2. We here assume that we are monitoring s that satisfies $s.w < (1-\epsilon)s^*.w$. Let c^* be the cell holding s^*, and we have that $s^*.w \leq s^*.\overline{w} \leq c^*.w$. Therefore, in this case, we definitely cannot prune OverlapComputation(c^*) and ExactWeightComputation(s^*, c^*) by Pruning rules 3 and 4, and s is replaced by s^*. This means that s cannot be the result, thus we conclude that Theorem 2 is true from the contradiction. □

6.2 Monitoring Top-k MaxRS

If the requirement is to monitor not only a single space but multiple spaces with the largest weight, a continuous top-k MaxRS query is a promising solution. This query achieves the requirement while controlling the result size as Definition 4 describes.

We know that Pruning rules 1 and 2 are based on a threshold, and the threshold is $s^*.w$, i.e., the (temporal) top-1 weight. It is intuitively known that the threshold is set as the kth largest weight in continuous top-k MaxRS queries.

Algorithm 6 illustrates the high level algorithm. Although the algorithm for continuous top-k MaxRS queries is essentially the same as Algorithm 2, the main modification is to deal with the set S^* of the k spaces with the maximum weight. We do not show OverlapComputation(C') and ExactWeightComputation(S^*, C') because they are also essentially the same as Algorithms 3 and 4, respectively.

Algorithm 6: Branch-and-bound algorithm using aG2 for continuous top-k MaxRS queries

1 Execute lines 1–5 in Algorithm 2
2 $C' \leftarrow \{\forall c_{i,j} \mid \exists s \in S^* \text{ is in } c_{i,j}\}$
3 **if** $C' = \emptyset$ *(all spaces in S^* expire)* **then**
4 　$C' \leftarrow \operatorname{argmax}_{c_{i,j} \in C} c_{i,j}.w$ // `C is the set of cells in aG2`
5 OverlapComputation(C')
6 $S^* \leftarrow$ ExactWeightComputation(S^*, C')
7 **for** $\forall c_{i,j} \in C \backslash C'$ **do**
8 　Execute lines 12–15 in Algorithm 2
9 **return** S^*

Table 4: Configuration of parameters

Parameter	Values
Window-size, n [$\times$1000]	100, 250, **500**, 750, 1000
Generation rate, m	50, **100**, 200, 500, 1000
Side length of a rectangle, l	100, 500, **1000**, 1500, 2000
Error rate, ϵ	0, 0.1, 0.2, 0.3, 0.4, 0.5
k	1, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50

7. EXPERIMENTS

This section provides our experimental results on the performances of our algorithms. Recall that this is the first work of monitoring MaxRS problem, so there is no existing algorithm that can deal with this problem. Therefore, to show the efficiency of the incremental approach of our algorithms, we use an algorithm with non-incremental approach as comparison. That is, we evaluated naive plane-sweep [12, 18], the algorithm using G2 (Section 4), and the branch-and-bound algorithms using aG2 (Sections 5 and 6). For easy recognition, our algorithms are represented by G2 and aG2. Recall that naive plane-sweep is an optimal in-memory algorithm for computing s^* from scratch, and the algorithm proposed in [8, 9] also uses naive plane-sweep in the case where all objects fit in the main memory. All algorithms were implemented in C++, and all experiments were conducted on a PC with 3.4GHz Intel Core i7 processor and 32GB RAM.

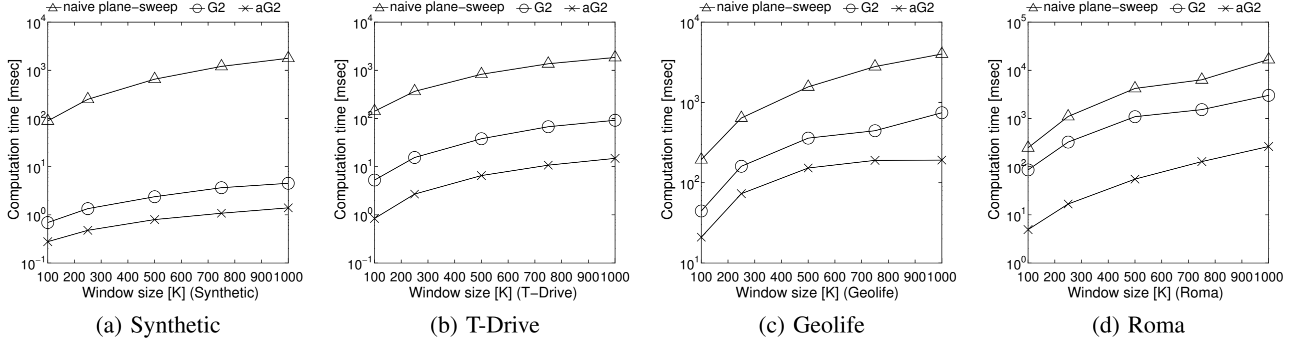
7.1 Setting

Datasets. We used a synthetic dataset and three real datasets. In the synthetic dataset, we generated objects under uniform distribution. The cardinality of this synthetic dataset is 10,000,000, and the range of each coordinate is [0, 1000000]. The three real datasets are T-Drive [31], Geolife [34], and Roma[2], which are the sets of continuously generated GPS data. The objects in the real datasets exist over a very wide range, thus we selected the objects existing around respective main areas. The cardinalities of T-Drive, Geolife, and Roma are 5,037,794, 3,662,876, and 8,368,858, respectively. The objects in the datasets are sorted in order of generation time, and we normalized the range of each coordinate to [0, 1000000]. In the above four datasets, the weight of a given object is a real-value randomly chosen from [0, 1000].

Parameters. Table 4 summarizes the parameters used in the experiments and bold values are default values. Note that a given rectangle is a square in the experiments, thus the size of a rectangle is $l \times l$, i.e., 1000×1000 by default.

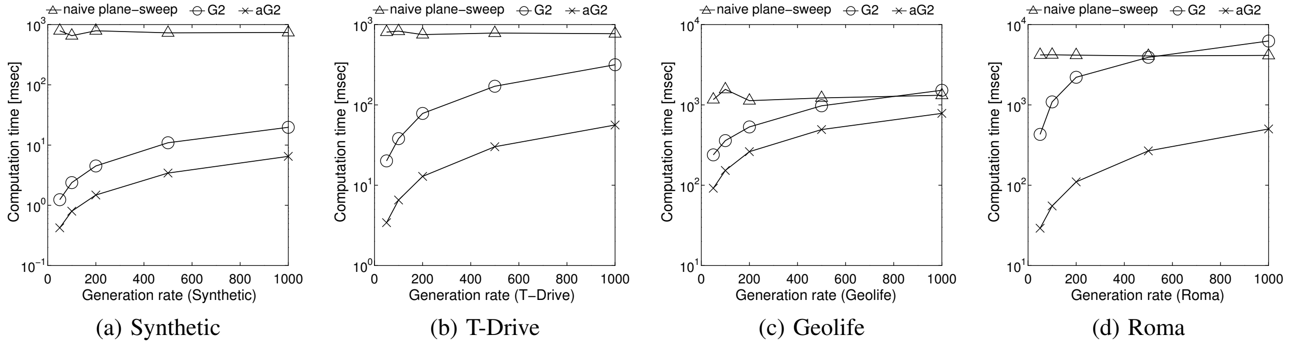
Evaluation. In Section 7.2, we investigate the impact of Algorithm 5. In Section 7.3, to investigate the performances of the algorithms w.r.t. monitoring MaxRS, we varied three parameters, n, m, and l,

[2]http://crawdad.org/index.html



(a) Synthetic (b) T-Drive (c) Geolife (d) Roma

Figure 7: Impact of n



(a) Synthetic (b) T-Drive (c) Geolife (d) Roma

Figure 8: Impact of m

Table 5: Computation time [msec]

Algorithm	Synthetic	T-Drive	Geolife	Roma
Algo. 2	0.799	6.547	152.644	55.196
Algo. 5 with cond.	0.796	5.952	202.127	49.245
Algo. 5	0.812	6.229	217.541	54.837

and measured the average computation time to update s^*. We also measured the practical error rate and computation time of aG2 by varying ϵ to investigate the performance of our approximate algorithm in Section 7.4. Let s be the space monitored by our approximate algorithm, and the practical error is defined by $1 - s.w/s^*.w$. Finally, we measured the average computation time to update S^* by varying k in Section 7.5.

7.2 Impact of Algorithm 5

We first study the impact of Algorithm 5 in our default setting. Table 5 shows the result. In Table 5, "Algo. 5 with cond." is denoted by Algorithm 2 with Algorithm 5 in the case where the upper-bounding cost is less than the plane-sweep algorithm. We can see that Algorithm 5 provides a trivial impact for all the four datasets and does not outperform Algorithm 2. We observed that $|R(r_i)|$ in Algorithm 5 is large in Geolife, thus Algorithm 5 provides a negative impact. Therefore we do not employ Algorithm 5 since it is not scalable .

7.3 Results on monitoring MaxRS

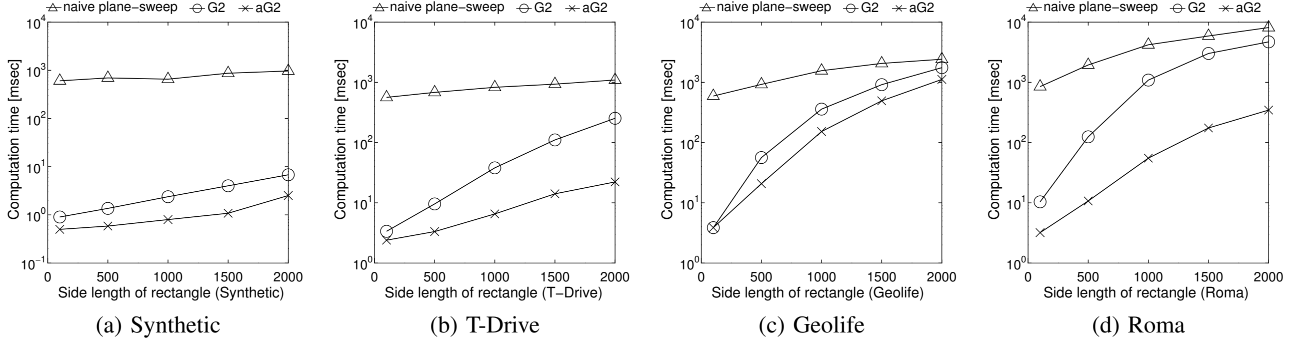
Impact of n. We study the impact of the number of objects on a sliding-window, and Figure 7 shows the results. All the algorithms need longer computation time as n increases. In terms of naive plane-sweep, this is intuitive since its cost is $\mathcal{O}(n \log n)$. It is also intuitive that the increase of the number of overlapped spaces due to the increase of n is likely to lose the chances which avoid OverlapComputation$(\cdot)$ and ExactWeightComputation$(\cdot, \cdot)$, thus our algorithms also need longer computation time. However, we can see that our algorithms are more efficient than naive plane-sweep in the four datasets, as expected. Naive plane-sweep is not scalable, which is shown in the case of large n. Moreover, aG2 scales better than G2 (the computation time is shown in log-scale). aG2 updates the result more than 2 times faster than G2 in the four datasets.

Impact of m. Next, we study the impact of the generation rate, i.e., the number of objects generated at the same time. Although the practical average generation rates of the three real datasets are less than 50 objects (per second), we employ larger generation rates to investigate the scalability of our algorithms. Figure 8 shows the results. Because naive plane-sweep computes s^* from scratch, it is not affected by m basically. The computation time of our algorithms increases as m increases. When m is large, the upper-bound weights maintained by cells and vertices are likely to become large, which results in the same observation as large n. Note that even a case of large m, e.g., $m = 1000$, we can observe that aG2 still updates the result faster than naive plane-sweep.

Impact of l. User-specified rectangle size also has impacts on the performances of continuous MaxRS query processing, because large rectangles tend to overlap with others. Figure 9 shows the results. We can see that aG2 keeps outperforming naive plane-sweep, but the tendencies are different between the datasets. In the uniform distribution, i.e., Figure 9(a), G2 and aG2 are not much affected by l, but in the real datasets, the computation time of our algorithms (and naive plane-sweep) increase as l increases. We observed that the distributions of the real datasets are *skewed*. Therefore many rectangles overlap with each other, then G2 and aG2 are likely to encounter the case that Local-Plane-Sweep$(\cdot)$ is not avoided.

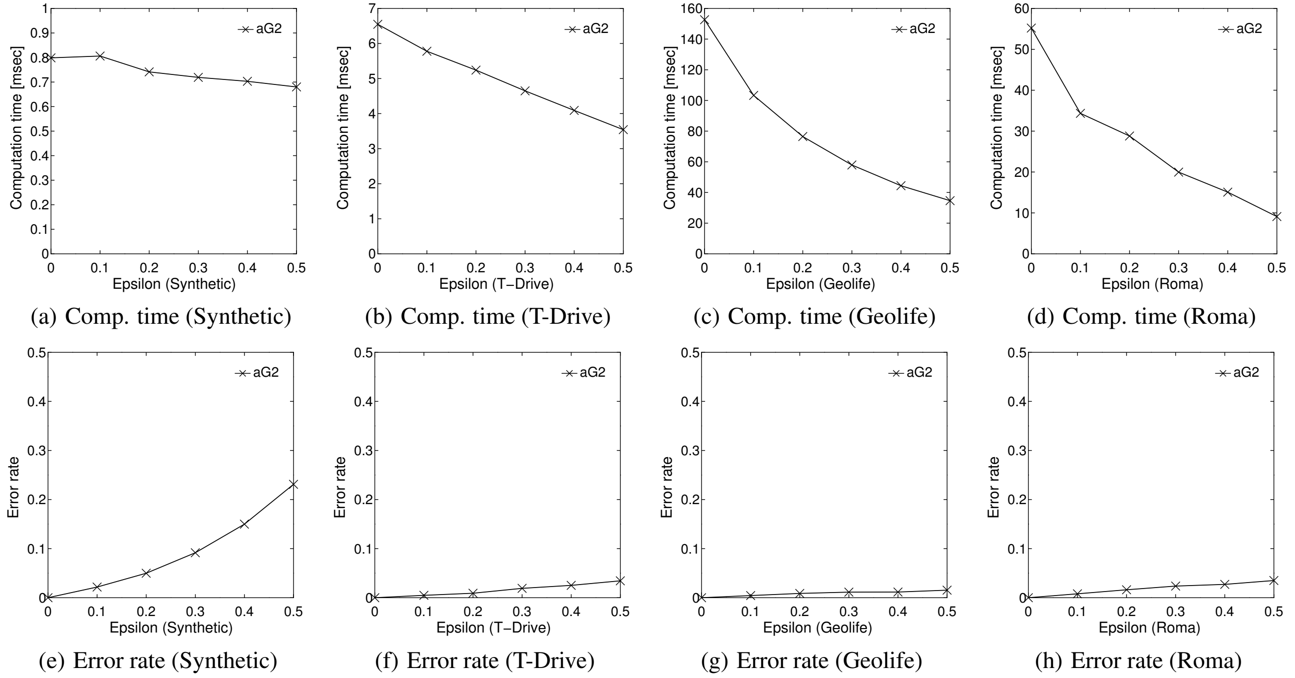
7.4 Results on monitoring Approximate MaxRS

We evaluated our approximate branch-and-bound algorithm us-



(a) Synthetic (b) T-Drive (c) Geolife (d) Roma

Figure 9: Impact of l



(a) Comp. time (Synthetic) (b) Comp. time (T-Drive) (c) Comp. time (Geolife) (d) Comp. time (Roma)

(e) Error rate (Synthetic) (f) Error rate (T-Drive) (g) Error rate (Geolife) (h) Error rate (Roma)

Figure 10: Impact of ϵ

ing aG2 by varying ϵ. Although an approximate algorithm for *one-time computation* of MaxRS queries has been proposed in [25], this algorithm cannot be used for comparison due to three reasons. First, this algorithm is based on randomized sampling, thus the answers returned by this algorithm and our algorithm are different. Second, the answer returned by this algorithm varies every time, which is not suitable as monitoring algorithm. Last, repeating such one-time computation is shown to be inefficient in Section 7.3. We therefore focus on our algorithm, and the experimental results are shown in Figure 10.

From Figures 10(a)–10(d), although its impact is different between the datasets, we can see that the computation time decreases as ϵ increases. This result satisfies the requirement that needs faster computation with an approximate answer. In the synthetic dataset, it seems that the computation time does not decrease so much (Figure 10(a)), but even when $\epsilon = 0$, the computation time is about 0.8 [msec], which is fast enough. From Figures 10(e)–10(h), we can see that as ϵ increases, the practical error also increases but is less than ϵ. The results show that the relationship between the query processing efficiency and the quality of the result is trade-off, but an interesting observation is that the practical error rates in the cases of the real datasets are very small.

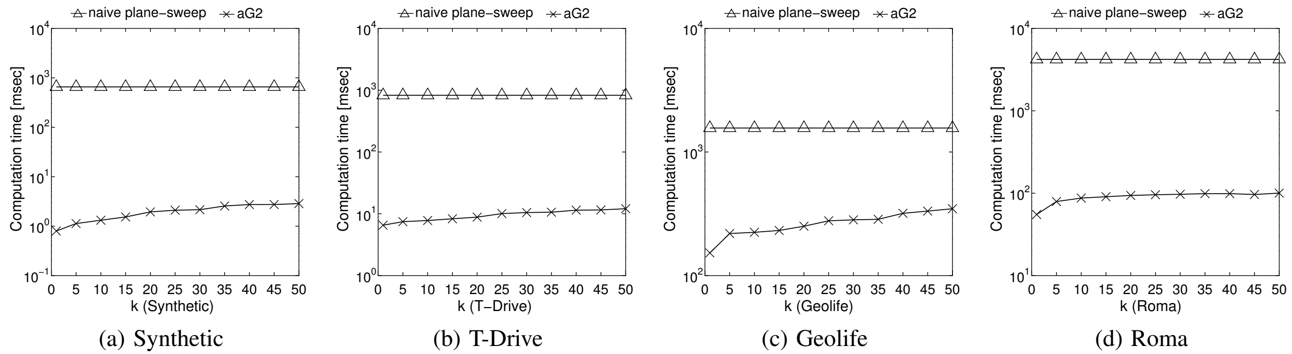
7.5 Results on monitoring Top-k MaxRS

To evaluate our branch-and-bound algorithm for continuous top-k MaxRS queries, we conducted an experiment by varying k. We compare our algorithm with naive plane-sweep. Although naive plane-sweep is for one-time computation, this algorithm can deal with top-k MaxRS queries without sacrificing the computation cost. We do not evaluate G2 since its performance is not better than aG2.

Figure 11 shows the results. Again, naive plane-sweep is not affected by k since it scans all rectangles on a sliding-window. As k increases, the computation time of aG2 increases, but we can see that the increase of the computation time of aG2 is slight for all the four datasets. These results confirm that the pruning rules keep efficient and avoid unnecessary computation.

8. CONCLUSION

In this paper, we addressed a novel problem of monitoring MaxRS and its variants, i.e., monitoring approximate MaxRS and top-k MaxRS. In the environments where spatio-temporal objects are generated frequently, monitoring and analysis of objects are often required, and a continuous MaxRS query is useful to support such requirements. Unfortunately, the existing solutions [8, 25] focus on static objects, thus are not efficient in our problem. Motivated



(a) Synthetic (b) T-Drive (c) Geolife (d) Roma

Figure 11: Impact of k

by this, we proposed the first algorithm that can incrementally update the exact result. This algorithm incorporates our index framework G2 (Graph in Grid index) which supports efficient query processing. We extended G2 and proposed aG2 (aggregate G2) and a branch-and-bound algorithm using aG2. The branch-and-bound algorithm accelerates the query processing efficiency. Moreover, we showed that the branch-and-bound algorithm can deal with the monitoring approximate MaxRS and the top-k MaxRS problems with simple modifications. To demonstrate the efficiency of our algorithms, we conducted experiments using synthetic and real datasets. The results show that the branch-and-bound algorithm using aG2 is superior to the one-time computation approach and the algorithm using G2.

As shown (but not theoretically) in Section 5.3, we need extra computation and storage costs to tight the upper-bound weights more. Hence, it is interesting to theoretically explore (or clarify the impossibility of) an approach that can tight the upper-bound the most without sacrificing the computational cost and storage cost. It is also interesting to develop an efficient algorithm that can deal with multiple continuous MaxRS queries at the same time. These are the works that need to be considered in the future.

Acknowledgment. This research is partially supported by the Grant-in-Aid for Scientific Research (A)(26240013) of MEXT, Japan, and JST, Strategic International Collaborative Research Program, SICORP.

9. REFERENCES

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[2] J. Bao, M. F. Mokbe, and C.-Y. Chow. Geofeed: A location aware news feed system. In *ICDE*, pages 54–65, 2012.

[3] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Continuous monitoring of distance-based range queries. *IEEE TKDE*, 23(8):1182–1199, 2011.

[4] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li. Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. *The VLDB Journal*, 21(1):69–95, 2012.

[5] L. Chen, G. Cong, X. Cao, and K.-L. Tan. Temporal spatial-keyword top-k publish/subscribe. In *ICDE*, pages 255–266, 2015.

[6] Z. Chen, Y. Liu, R. C.-W. Wong, J. Xiong, X. Cheng, and P. Chen. Rotating maxrs queries. *Information Sciences, Elsevier*, 305:110–129, 2015.

[7] Z. Chen, Y. Liu, R. C.-W. Wong, J. Xiong, G. Mai, and C. Long. Efficient algorithms for optimal location queries in road networks. In *SIGMOD*, pages 123–134, 2014.

[8] D.-W. Choi, C.-W. Chung, and Y. Tao. A scalable algorithm for maximizing range sum in spatial databases. *PVLDB*, 5(11):1088–1099, 2012.

[9] D.-W. Choi, C.-W. Chung, and Y. Tao. Maximizing range sum in external memory. *ACM TODS*, 39(3):21, 2014.

[10] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *SSTD*, pages 163–180. 2005.

[11] J. Huang, Z. Wen, J. Qi, R. Zhang, J. Chen, and Z. He. Top-k most influential locations selection. In *CIKM*, pages 2377–2380, 2011.

[12] H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of algorithms, Elsevier*, 4(4):310–323, 1983.

[13] C. Li, Y. Gu, J. Qi, G. Yu, R. Zhang, and W. Yi. Processing moving *k*nn queries using influential neighbor sets. *PVLDB*, 8(2):113–124, 2014.

[14] Y. Liu, R. C.-W. Wong, K. Wang, Z. Li, C. Chen, and Z. Chen. A new approach for maximizing bichromatic reverse nearest neighbor search. *Knowledge and Information Systems, Springer*, 36(1):23–58, 2013.

[15] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of spatial queries in wireless broadcast environments. *IEEE TMC*, 8(10):1297–1311, 2009.

[16] K. Mouratidis and D. Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE TKDE*, 19(6):789–803, 2007.

[17] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.

[18] S. C. Nandy and B. B. Bhattacharya. A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. *Computers & Mathematics with Applications, Elsevier*, 29(8):45–61, 1995.

[19] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *SSTD*, pages 443–459. 2001.

[20] J. Qi, R. Zhang, L. Kulik, D. Lin, and Y. Xue. The min-dist location selection query. In *ICDE*, pages 366–377, 2012.

[21] S. Rahul and Y. Tao. On top-k range reporting in 2d space. In *PODS*, pages 265–275, 2015.

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

[23] C. Sheng and Y. Tao. New results on two-dimensional orthogonal range aggregation in external memory. In *PODS*, pages 129–139, 2011.

[24] Y. Sun, J. Qi, Y. Zheng, and R. Zhang. K-nearest neighbor temporal aggregate queries. In *EDBT*, pages 493–504, 2015.

[25] Y. Tao, X. Hu, D.-W. Choi, and C.-W. Chung. Approximate maxrs in spatial databases. *PVLDB*, 6(13):1546–1557, 2013.

[26] Y. Tao, C. Sheng, C.-W. Chung, and J.-R. Lee. Range aggregation with set selection. *IEEE TKDE*, 26(5):1240–1252, 2014.

[27] S.-H. Wu, K.-T. Chuang, C.-M. Chen, and M.-S. Chen. Diknn: an itinerary-based knn query processing algorithm for mobile sensor networks. In *ICDE*, pages 456–465, 2007.

[28] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On computing top-t most influential spatial sites. In *VLDB*, pages 946–957, 2005.

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

[30] M. L. Yiu and N. Mamoulis. Multi-dimensional top-k dominating queries. *The VLDB Journal*, 18(3):695–718, 2009.

[31] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *SIGKDD*, pages 316–324, 2011.

[32] D. Zhang, Y. Du, T. Xia, and Y. Tao. Progressive computation of the min-dist optimal-location query. In *VLDB*, pages 643–654. t, 2006.

[33] J. Zhang, W.-S. Ku, M.-T. Sun, X. Qin, and H. Lu. Multi-criteria optimal location query with overlapping voronoi diagrams. In *EDBT*, pages 391–402, 2014.

[34] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from gps trajectories. In *WWW*, pages 791–800, 2009.

[35] Z. Zhou, W. Wu, X. Li, M. L. Lee, and W. Hsu. Maxfirst for maxbrknn. In *ICDE*, pages 828–839, 2011.